UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UMA TÉCNICA BASEADA EM SysML PARA MODELAR A ARQUITETURA DE SISTEMAS EMBARCADOS DE TEMPO REAL

Quelita Araújo Diniz da Silva Ribeiro

São Cristóvão

2017

UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E TECNOLÓGICAS

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Quelita Araújo Diniz da Silva Ribeiro

UMA TÉCNICA BASEADA EM SysML PARA MODELAR A ARQUITETURA DE SISTEMAS EMBARCADOS DE TEMPO REAL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Michel dos Santos Soares

São Cristóvão
2017

Quelita Araújo Diniz da Silva Ribeiro

UMA TÉCNICA BASEADA EM SysML PARA MODELAR A ARQUITETURA DE SISTEMAS EMBARCADOS DE TEMPO REAL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação (PROCC) da Universidade Federal de Sergipe (UFS) como parte de requisito para obtenção do título de Mestre em Ciência da Computação.

BANCA EXAMINADORA

Prof. Dr. Michel dos Santos Soares, Presidente
Universidade Federal de Sergipe (UFS)

Prof. Dr. Rogério Patrício Chagas do Nascimento, Membro
Universidade Federal de Sergipe (UFS)

Prof. Dr. Pedro Frosi Rosa, Membro
Universidade Federal de Uberlândia (UFU)

UNIVERSIDADE FEDERAL DE SERGIPE
PRÓ-REITORIA DE PÓS-GRADUAÇÂO E PESQUISA
NÚCLEO DE PÓS-GRADUAÇÂO EM CIÊNCIA DA COMPUTAÇÃO

Relatório de defesa pública do (a) Senhor(a) **QUELITA ARAUJO DINIZ DA SILVA RIBEIRO** no Programa de Ciência da Computação (PROCC) da UFS.

Aos 23 dias do mês de março de 2017, realizou-se a **Defesa de Mestrado** do trabalho intitulado **"UMA TÉCNICA BASEADA EM SysML PARA MODELAR A ARQUITETURA DE SISTEMAS EMBARCADOS DE TEMPO REAL"** sob orientação do Profº. Drº. **Michel dos Santos Soares.**

Depois de declarada aberta a sessão, o Presidente da Banca passou inicialmente a palavra ao candidato para exposição e a seguir aos examinadores para as devidas arguições que se desenvolveram nos termos regimentais. Em seguida, a comissão julgadora proclamou o resultado:

Nome Banca Examinadora	Instituição	Nota (de 0 a 10)	Assinatura
Michel dos Santos Soares	UFS		Michel S. Soares
Rogerio Patricio Chagas Nascimento	UFS		
Pedro Frosi Rosa	UFU		
	Média =		

Dessa maneira o Resultado Final é: **APROVADO** ou **REPROVADO**

Parecer da Banca Examinadora *

- Obs: Se o candidato for reprovado, o preenchimento do parecer é obrigatório.

São Cristóvão/SE

Michel S. Soares
Assinatura do Orientador:

Quelita A. D. da S. Ribeiro
Assinatura do Aluno:

Agradecimentos

Agradeço primeiramente a Deus, meu pai e amigo. Creio fielmente que esteve ao meu lado em todos os momentos de minha vida e, por isso, O amo tanto! E sou grata por esse sonho realizado.

A meu orientador Michel, professor e amigo, obrigada pelo direcionamento durante a realização deste trabalho. A sua dedicação, apoio, paciência, sabedoria e carinho foram extremamente relevantes para mim.

A minha mãe Dacylete, como eu te amo mainha! Eu vou sempre ser grata porque a senhora é tudo na minha vida! Eu te amo, te amo, te amo...

A minha família: Edivaldo, Giuvanice, Dacylete, Kaetillyn e Diego, pela força, motivação, paciência (muita paciência), carinho, compreensão, amor e apoio. Sempre presentes, incentivadores e pacientes. Eu amo muito todos vocês.

Ao querido amigo Luciano, obrigada pela ajuda, conselhos, incentivo, força, ânimo e por acreditar em mim!

Aos amigos e colegas de mestrado Jamille e Telmo que compartilharam comigo essa conquista, companhia, risadas e, muitas vezes, o dia-a-dia. Obrigada pela amizade!

Aos meus amigos e colegas, pelo companheirismo e vibração por esta jornada. Principalmente a Davi, Chaina, Íris, as Elaines, Aninha, Dina e Erickson, as “azamigas da onça” Isabel, Raisa e Marianna. Ah e a Glauber, que ainda me deve um *petit gateau*!!

Aos professores, que compartilharam os seus conhecimentos comigo. Principalmente, aos professores Rogério Nascimento e Pedro Frosi que participaram da banca de defesa deste trabalho.

A CAPES pelo apoio financeiro em todo o período do mestrado.

Aos meus parentes que torceram muito pelo meu sucesso sempre fervorosos em minha conquista.

A todos que colaboraram comigo para que este trabalho fosse realizado. Meu crescimento educacional e profissional não seria igual se não os tivesse ao meu lado nos melhores e piores momentos.

Resumo

A especificação da arquitetura de sistemas de software de tempo real é uma atividade que depreende análise, conhecimento e compreensão tanto do domínio da aplicação quanto das partes envolvidas na construção do software. A arquitetura tem um papel primordial na comunicação entre os *stakeholders*, além do planejamento de todo o processo arquitetural envolvido no projeto. Contudo, as Linguagens de Descrição de Arquiteturas (ADLs) não têm sido amplamente usadas na indústria. Outro fator limitador para o uso efetivo de ADLs é a dificuldade dessas linguagens em expressar efetivamente a arquitetura de sistemas complexos. Considerando essa situação de dificuldade do uso efetivo de ADLs, a UML tem sido utilizada nos últimos anos para modelagem da arquitetura. No entanto, a UML não consegue representar características importantes pertinentes a sistemas de tempo real, tais como segurança ou restrições de tempo real. Uma das vantagens da UML é a capacidade de extensão permitindo a criação de *profiles*. Neste sentido, este trabalho apresenta a *Systems Modeling Language* (SysML), um *profile* da UML, para modelagem da arquitetura de sistemas de tempo real em dois sistemas automotivos, o sistema de controle de *airbag* e o sistema de controle de faróis. Neste trabalho tem-se como objetivos utilizar a UML e a SysML para modelagem e documentação da arquitetura e delineamento de rastreabilidade de requisitos entre software e sistema, ampliando o entendimento do projeto entre as partes envolvidas, e por fim apresentar a SysML como uma linguagem para descrição da arquitetura de software de tempo real. As linguagens SysML e a ADL *Architecture Analysis & Design Language* (AADL) foram comparadas para mostrar as vantagens da SysML. Como resultado, foi percebido que características abstratas, tais como tomadas de decisão, repetição de uma funcionalidade (*loop*), características que são relacionadas a realidade e, consequentemente, ao sistema, não podem ser descritas em AADL. A SysML mostrou-se relevante no contexto da descrição, análise, classificação e modelagem de arquitetura para sistemas de tempo real. O diagrama de Requisitos da SysML mostra explicitamente os diversos tipos de relacionamentos entre diferentes requisitos, o diagrama de Blocos viabiliza a visão global dos sistemas envolvidos num único projeto, o diagrama de Blocos Internos possibilita a visão interna do sistema em construção, o diagrama de Atividades considera a visão comportamental do sistema. Os conceitos de SysML, articulados nos diagramas de Requisitos, Atividades, Blocos e Blocos Internos da SysML são complementares cobrindo os propósitos necessários para a descrição da arquitetura de sistemas de tempo real. Conclui-se que a técnica proposta da junção de UML e SysML fornece elementos para descrever requisitos de software e seus relacionamentos com o sistema, gerenciar mudanças, evoluir e rastrear requisitos mais facilmente, além da comunicação ser efetivamente realizada entre os *stakeholders*. Este aspecto é importante ao desenvolvimento de sistemas de tempo real, por causa da diversidade de pessoas/equipes envolvidas e que influenciam uma ampla série de decisões de projeto.

Palavras-chaves: SysML, Arquitetura de Software, ADL, AADL, UML.

Abstract

Architectural specification of real-time software systems is an activity that conveys analysis, knowledge and understanding of both the application domain and the parties involved in software construction. Architecture plays a key role in communication between stakeholders, in addition to planning the entire architectural process involved in the project. However, Architecture Description Languages (ADLs) have not been widely used in the industry. Another limiting factor for the effective use of ADLs is the difficulty of these languages in effectively expressing the architecture of complex systems. Considering this situation of difficulty in the effective use of ADLs, the UML has been used in recent years to model the architecture. However, UML can not represent the important characteristics pertinent to real-time systems, such as security or real-time constraints. One of the advantages of the UML is the extensibility allowing the creation of profiles. In this sense, this work proposes using Systems Modeling Language (SysML), a UML profile, to model real-time systems architecture in two automotive systems, the airbag control system and the light control system. The objective of this work is to use UML and SysML to model and document the architecture and design of requirements traceability between software and systems elements, increasing the understanding of the project among the parties involved, and finally presenting SysML as a language for description of real-time software architecture. The ADL Architecture Analysis and Design Language (AADL) and SysML languages were compared to show the advantages of SysML. As a result, it was noticed that abstract features such as conditional deviations, loop, characteristics that are related to reality and consequently to the system can not be described in AADL. SysML has proved to be relevant in the context of architecture description, analysis, classification and modeling of real-time systems. The SysML Requirements diagram explicitly shows the various types of relationships between different requirements, the Block diagram enables the global view of the systems involved in a single project, the Internal Block diagram allows the internal view of the system under construction, the Activity diagram considers the behavioral view of the system. SysML concepts, articulated in the SysML Requirements, Activities, Blocks and Internal Blocks diagrams, are complementary, covering the purposes needed to describe the architecture of real-time systems. It is concluded that the proposed UML and SysML join technique provides elements to describe software requirements and their relationships with the system, to manage changes, to evolve and to trace requirements more easily, in addition to the communication being effectively carried out between the stakeholders. This is important for the development of real-time systems because of the diversity of people/teams involved and influencing a wide range of design decisions.

Key-words: SysML, Software Architecture, ADL, AADL, UML.

Lista de figuras

Lista de tabelas

Lista de abreviaturas e siglas

AADL	Architecture Analysis & Design Language
ADL	Architecture Description Language
AOP	Aspect-oriented Programming
BDD	Block Definition Diagram
CCSL	Clock Constraint Specification Language
CMU	Carnegie Mellon University
ECU	Engine Control Unit
IEC	International Electrotechnical Commission
IBD	Internal Block Diagram
ICEIS	International Conference on Enterprise Information Systems
INCOSE	International Council on Systems Engineering
ISO	International Organization for Standardization
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group
PKU	Peking University, Beijing
SAE	Society of Automotive Engineers
SysML	Systems Modeling Language
SOP	Subject-oriented Programming
SRS	Software Requirements Specification
UCI	University of California, Irvine
UML	Unified Modeling Language
UPV	Universitat Politècnica de València
XML	Extensible Markup Language

Sumário

1 Introdução

Com o aumento em tamanho e complexidade dos sistemas de software, o problema do projeto e desenvolvimento de software vai além dos algoritmos e estruturas de dados (GARLAN; SHAW, 1994), envolvendo também fortemente a concepção e especificação arquitetural do software. A arquitetura do software está relacionada com a estrutura e organização do software, interligação de protocolos de comunicação, sincronização e acesso a dados, distribuição física, composição dos elementos de *design*, escalabilidade, desempenho e seleção entre as alternativas de projeto (GARLAN; SHAW, 1994).

Os requisitos de software estão altamente relacionados com a arquitetura de software. Os requisitos capturam uma descrição do sistema (AUWERAER et al., 2013). Elicitar e capturar os requisitos de sistemas de software complexos de forma consistente é uma tarefa desafiadora, exigindo que tanto os requisitos como os modelos estejam bem alinhados e consistentes uns com os outros (CHIAO; KUNZLE; REICHERT, 2013). A Norma ISO/IEC/IEEE 29148:2011 reúne práticas recomendadas para a especificação de requisitos de projeto de sistemas (ISO, 2011). Um ciclo típico de projeto de software inicia com a elicitação de requisitos. Uma vez que os requisitos são definidos, os modelos podem ser criados e a arquitetura do software pode ser definida (AUWERAER et al., 2013).

A arquitetura de software é essencial durante todas as fases de desenvolvimento do software, tendo em vista que a descrição da arquitetura afeta o sucesso de um projeto e ajuda os *stakeholders* a compreenderem facilmente o software (GARDAZI et al., 2009). Nesse contexto, também é objetivo da arquitetura de software ser um veículo de comunicação entre os *stakeholders* (PANDEY, 2010).

A arquitetura de software emergiu como o estudo da estrutura geral do software, especialmente as relações entre subsistemas e componentes (SHAW, 2001). A especificação arquitetural serve para analisar e descrever propriedades de um sistema complexo, permitindo ao arquiteto ter uma visão completa do sistema (GARLAN; SHAW, 1994). As responsabilidades dos arquitetos são conceber e tomar decisões no projeto que tenham impactos duradouros nos atributos de qualidade de um software, como evolução, desempenho e segurança (KRUCHTEN, 2008). As decisões do arquiteto baseiam-se em definir e manter a arquitetura do software, avaliar e mitigar os riscos, planejar e resolver problemas do projeto (KRUCHTEN, 2008).

A modelagem da arquitetura é desenvolvida por meio de notações gráficas que são chamadas de *Architecture Description Language* (ADL) (MEDVIDOVIC; TAYLOR, 2000). Muitas ADLs foram propostas por pesquisadores e profissionais para aplicações em domínios gerais ou específicos (MEDVIDOVIC; TAYLOR, 2000), (MALAVOLTA et al., 2013). Porém, ADLs como C2, RAPIDE, Darwin, Wright, ACME e UniCon são pouco usadas na indústria

de software (KRUCHTEN; STAFFORD, 2006), (OZKAYA; KLOUKINAS, 2013), (ALMARI; BOUGHTON, 2014). As razões pelas quais as ADLs são pouco utilizadas na prática incluem sua difícil manipulação em geral (OZKAYA; KLOUKINAS, 2013), (MALAVOLTA et al., 2013), o fato da maioria das ADLs terem sido criadas para aplicações de domínio específico, como as aplicações aéreas (PANDEY, 2010), e ainda porque poucas ADLs têm suporte de boas ferramentas dificultando a aplicação na prática (PANDEY, 2010). Além disso, foram evidenciadas limitações nas ADLs, incluindo falta de apoio adequado a múltiplas visões (HILLIARD; RICE, 1998), (PANDEY, 2010), falta de recursos para documentar as decisões do arquiteto (HILLIARD; RICE, 1998), (PANDEY, 2010), falta de apoio adequado para restrições (HILLIARD; RICE, 1998), (PANDEY, 2010), estilos e padrões arquiteturais (HILLIARD; RICE, 1998). Em consequência, têm-se iniciativas que usam a *Unified Modeling Language* (UML) para descrever graficamente a arquitetura. Estima-se que cerca de 86% da indústria de software usa UML ou algum *profile* da UML para descrever a arquitetura de software (MALAVOLTA et al., 2013).

Diversos *profiles* da UML foram propostos para melhorar a modelagem da arquitetura e superar as limitações existentes nas ADLs (MALAVOLTA et al., 2013). Segundo Malavolta et al. (2013), muitas propostas foram apresentadas para modelar a arquitetura e estender a UML 1.x, dentre elas (ROBBINS et al., 1998), (GARLAN; KOMPANEK, 2000), (GOMAA; WIJESEKERA, 2001), (SELIC, 2001), (MEDVIDOVIC et al., 2002) e (KANDÉ et al., 2002). Em seguida outras propostas para estender a UML 2.x foram publicadas (GOULÃO; ABREU, 2003), (ROH; KIM; JEON, 2004) e (PÉREZ-MARTÍNEZ; SIERRA-ALONSO, 2004). Mas a UML não foi desenvolvida com o objetivo de modelar a arquitetura de software e não converge para um padrão e notação de modelagem arquitetural (MALAVOLTA et al., 2013). A UML carece de semântica formal (PANDEY, 2010), (OZKAYA; KLOUKINAS, 2013), não é adequada para análise automatizada de verificação e validação da arquitetura (PANDEY, 2010) e não oferece suporte a especificação de conectores[1] (OZKAYA; KLOUKINAS, 2013). Todavia, a UML se tornou popular na indústria de software como uma linguagem de modelagem arquitetural (PANDEY, 2010).

Neste trabalho é proposto usar a SysML *(Systems Modeling Language)* como uma linguagem para modelagem da arquitetura de software. SysML é um *profile* da UML 2.0 para modelagem de propósito geral aplicado a sistemas (OMG, 2015). Os desafios encontrados na SysML para modelagem da arquitetura de software são analisados e este trabalho apresenta soluções para estender a SysML com elementos de modelagem de arquitetura de software.

1.1 Objetivos

Diante da importância em considerar a modelagem da arquitetura de software, este trabalho tem o objetivo geral de avaliar a SysML como uma linguagem para modelagem de

[1] Conectores representam relações ou interações entre componentes (PETTY; MCKENZIE; XU, 2002).

arquitetura de software. Os objetivos específicos deste trabalho são:

- Especificar requisitos dos sistemas de controle de *airbag* e de faróis de acordo com a Norma ISO/IEC/IEEE 29148:2011. Este objetivo específico é tratado no Capítulo 3;
- Definir critérios baseados em estudos da literatura para comparar ADLs. Este objetivo específico é tratado no Capítulo 4;
- Identificar, analisar e comparar estudos a respeito das linguagens de descrição da arquitetura de software. Este objetivo específico é tratado no Capítulo 4;
- Utilizar a SysML como linguagem de modelagem de arquitetura de software em uma aplicação. Este objetivo específico é tratado no Capítulo 5;
- Combinar a UML e a SysML para modelagem da arquitetura de sistemas e de arquitetura de software. Este objetivo específico é tratado no Capítulo 5;
- Comparar a SysML com a AADL. Este objetivo específico é tratado no Capítulo 6;
- Apresentar os benefícios, dificuldades, vantagens e desvantagens encontradas na utilização da SysML como linguagem de modelagem de arquitetura de software. Este objetivo específico é tratado nos Capítulos 6 e 7.

1.2 Revisão de Literatura

Com o objetivo de comparar as ADLs com UML, no artigo Pandey (2010) são apresentados, genericamente, os pontos fortes e fracos das ADLs e da UML. O autor explica os motivos das ADLs serem pouco usadas na indústria de software e também afirma que a UML é usada na indústria de software não apenas para projeto de software orientado a objetos mas também para a modelagem de arquitetura de software. O artigo contém uma breve discussão sobre a UML ser ou não considerada uma ADL. O autor expõe que a UML não é reconhecida como uma ADL, mas tornou-se uma notação padrão de fato para documentar a arquitetura de sistemas de software.

Em Ozkaya e Kloukinas (2013) é destacado que as pesquisas em arquitetura de software vêm evoluindo desde 1990. Como consequência, as ADLs descrevem o projeto arquitetural desde o início do desenvolvimento de um software e vêm facilitando a comunicação entre os *stakeholders*. Para os autores, após esse período de pesquisas era esperado que um número razoável de ADLs fossem escolhidas por profissionais na indústria de software. No entanto, as ADLs tiveram relativa baixa aceitação na indústria de software, quando comparadas com linguagens para modelagem de projeto de software. Os autores acreditam que o problema desse fato é que as ADLs são difíceis de usar e uma pequena quantidade possui semântica formal. O artigo contém uma análise comparativa dos problemas de treze (13) ADLs. A análise das ADLs

serve de base para a construção de XCD, uma nova ADL, que possibilita usabilidade para os profissionais.

Na pesquisa de Malavolta et al. (2013) é apresentada uma lacuna existente entre as tecnologias das ADLs e o que é necessário para os usuários das ADLs. As linguagens foram desenvolvidas em pesquisas acadêmicas e por profissionais na indústria de software nas últimas duas décadas. No entanto, não está claro se as linguagens satisfazem às necessidades dos *stakeholders*. Com o objetivo de auxiliar a concepção da próxima geração de linguagens, os autores analisam pontos fortes, limitações das linguagens e necessidades dos profissionais para modelar a arquitetura de software na indústria mediante questionário e entrevistas. Na análise dos dados, os autores concluíram que os profissionais estão satisfeitos com as capacidades das linguagens que usam, mas estão insatisfeitos com os recursos de análise da linguagem.

No estudo de Almari e Boughton (2014) é destacado que o desenvolvimento de software tem resultados melhores quando possui descrições precisas da arquitetura. Os autores citam que a variedade de artefatos existentes é um problema. Outro problema retratado na pesquisa é que a maioria dos desenvolvedores não costuma modelar o software. Os autores investigaram cinco fatores que incentivam a avaliação da arquitetura de software que são formalidade nas descrições, modelagem, documentação, padronização e instrumentos de avaliação da arquitetura. Na pesquisa é evidenciada a importância em encontrar soluções para os obstáculos do uso de ADLs na descrição da arquitetura de software.

Em Pandey (2010) é retratado que houve divergências entre os pesquisadores em considerar UML como uma ADL, porém a UML possui mais recursos que são incorporados em novas versões e *profiles*. Este é um dos motivos deste trabalho propor a SysML como uma linguagem para modelagem de arquitetura de software. Além disso, neste trabalho é proposto usar e adaptar a SysML para avaliar quais vantagens e desvantagens a SysML apresenta na modelagem da arquitetura, em vez de sugerir a criação de uma nova ADL como no trabalho de Ozkaya e Kloukinas (2013), o que pode implicar em grandes alterações na metodologia de desenvolvimento de software da organização e altos custos com novas ferramentas e treinamentos, dificultando a adoção da ADL na prática.

1.3 Metodologia

Para que os objetivos deste trabalho sejam atingidos, três instrumentos de pesquisa foram aplicados: a revisão bibliográfica, a análise comparativa e o estudo de caso.

Neste trabalho, os artigos científicos foram consultados nas bibliotecas digitais *ACM, IEEExplore, ScienceDirect e Scopus* nos últimos cinco anos. O intuito da revisão bibliográfica foi encontrar trabalhos relevantes que auxiliaram a construir o conhecimento sobre os problemas, dificuldades e desafios da modelagem da arquitetura de software e das ADLs. Pretendeu-se com isso catalogar quais as principais vantagens, desvantagens e desafios de 10 ADLs e mostrar onde

é possível avançar nessa área. Em seguida, comparar a SysML com critérios definidos a partir de outras ADLs.

Neste trabalho, 16 características foram especificadas como critérios de avaliação para a análise comparativa entre as ADLs. Todas as características são base para uma análise comparativa entre as ADLs AADL, ABC/ADL, C2, COSA, Darwin, PRISMA, RAPIDE, SOFA, UniCon e Wright.

O estudo de caso é considerado um instrumento de pesquisa adequado para a pesquisa em engenharia de software (RUNESON; HÖST, 2009), pois serve para que o pesquisador possa constatar que a teoria proposta é adequada no ambiente real (TRAVASSOS; GUROV; AMARAL, 2002). De acordo com Shaw (2002), a pesquisa em engenharia de software pode ser caracterizada em termos de problemas práticos, e os objetivos-chave da pesquisa são qualidade, custo e oportunidade de produtos de software. Neste trabalho foram apresentados, como estudos de caso, a modelagem de dois cenários de arquitetura de sistemas embarcados de tempo real com SysML, em um deles foi evidenciada uma nova técnica arquitetural e no outro cenário foi realizada a comparação com as linguagens SysML e AADL.

Shaw (2002) classifica os tipos de resultados e validação em pesquisas científicas. O resultado deste trabalho é uma técnica (apresentada no Capítulo 5), pois foi produzida uma maneira de projetar uma arquitetura de software com ênfase na comunicação de *stakeholders* e rastreabilidade de requisitos de software. Este trabalho é motivado pela realidade porque nele são apresentados dois exemplos, sistema de controle de *airbag* e sistema de controle de faróis. As descrições e os conceitos a respeito do sistema veicular foram consultados no livro (ZURAWSKI, 2009).

1.4 Visão Geral da Dissertação

Este trabalho contribui para a pesquisa e prática em Engenharia de Software, especialmente para a subárea da Arquitetura de Software. É proposta também uma extensão de profile SysML com sua integração a diagramas de Classes da UML. Foram desenvolvidos dois projetos arquiteturais de sistemas automotivos embarcados de tempo real em dois estudos de caso.

O texto desta dissertação está organizado nos seguintes Capítulos. No Capítulo 2, é realizada a fundamentação teórica sobre a pesquisa proposta descrevendo conceitos da arquitetura de software, linguagem de descrição de arquitetura, SysML, sistemas embarcados e sistemas embarcados automotivos. No Capítulo 3, os requisitos relacionados aos sistemas de controle de *airbag* e de faróis relevantes a aplicação dos estudos de caso, apresentados nos capítulos posteriores, são mostrados. A análise comparativa das ADLs está no Capítulo 4. A modelagem arquitetural do sistema de controle de *airbag* está descrita no Capítulo 5. Além da modelagem, uma técnica proposta combinando SysML e UML é explicada, com o objetivo de melhorar a comunicação dos *stakeholders*, identificar e ampliar as visões arquiteturais, principalmente

ao que se refere a software e sistema. O foco principal no Capítulo 6 é identificar, analisar e apresentar os resultados obtidos da comparação entre as linguagens AADL e SysML em um estudo de caso. Por fim, no Capítulo 7 são apresentadas as conclusões referentes ao trabalho e propostas que viabilizam trabalhos e pesquisas futuras.

2 Referencial Teórico

Neste capítulo são abordados conceitos fundamentais relacionados a arquitetura de software, linguagens de descrição da arquitetura de software, linguagem SysML, sistemas embarcados e sistemas automotivos. Na Seção 2.1 são apresentadas definições a respeito da arquitetura de software, objetivos e vantagens do desenvolvimento do projeto arquitetural. Conceitos relevantes relacionados às linguagens de descrição da arquitetura de software são descritos na Seção 2.2. A linguagem SysML é abordada na Seção 2.3, juntamente com os diagramas que compõem a linguagem. Os conceitos pertinentes aos sistemas embarcados e sistemas automotivos são apresentados nas Seções 2.4 e 2.5, respectivamente.

2.1 Arquitetura de Software

Provavelmente, a primeira referência à arquitetura de software ocorreu em 1969 em uma conferência sobre engenharia de software (KRUCHTEN; STAFFORD, 2006). Porém, o conceito da arquitetura de software, como uma disciplina específica da Engenharia de Software, obteve maior atenção da comunidade a partir da década de 1990. Um artigo produzido por Winston Royce e Walker Royce em 1991 foi o primeiro a posicionar arquitetura de software no título de um artigo (ROYCE; ROYCE, 1991), (KRUCHTEN; STAFFORD, 2006). Em 1992, Perry e Wolf (1992) publicaram um artigo e nele introduziram a definição da arquitetura de software consistindo na equação:

$$\{\text{Elementos, Organização, Decisões}\} = \text{Arquitetura de software},$$

De acordo com essa definição, a arquitetura de software é um conjunto de elementos arquiteturais que possuem uma organização. Os elementos e sua organização são definidos por decisões tomadas pelo arquiteto para atender objetivos e restrições (PERRY; WOLF, 1992). Essa equação foi base para a criação de ADLs tais como C2, RAPIDE, Darwin, Wright, ACME, e UniCon (KRUCHTEN; STAFFORD, 2006).

Atualmente, ainda não há um consenso na comunidade científica sobre a definição da arquitetura de software. Por esse motivo, existem tentativas e divergências em definir o termo (KRUCHTEN; STAFFORD, 2006).

Outra definição provém da ISO:IEC:IEEE:42010 (2011), que define a arquitetura de software como uma organização fundamental de um sistema incluindo componentes, relacionamentos com o ambiente e os princípios que conduz a concepção e evolução de um sistema de software.

A descrição da arquitetura é usada para expressar uma arquitetura de um projeto de um sistema de software (ISO, 2011). Arquitetura refere-se a conceitos fundamentais ou propriedades de um sistema em seu ambiente incorporado em seus elementos, relações e nos princípios de seu projeto e evolução (ISO, 2011).

Além de conceber e especificar a estrutura geral do software, a arquitetura de software é uma parte reconhecida e indispensável ao longo do desenvolvimento de um software (VLIET, 2008). Ela consiste na captura das principais decisões de modelagem. Estas decisões de projeto são importantes uma vez que suas ramificações são sentidas em todas as fases subsequentes da construção de um software (VLIET, 2008). Nesse contexto, reunir os conjuntos de decisões tomadas pelo arquiteto consiste em documentar a arquitetura (VLIET, 2008). Com relação a isso, as vantagens nas descrições da arquitetura de software são percebidas em atributos de qualidade (requisitos funcionais e não funcionais), evolução, reusabilidade, redução da complexidade do software e aprimoramento da comunicação entre os *stakeholders* (BASS; CLEMENTS; KAZMAN, 2012).

A visão arquitetural expressa a arquitetura de um sistema na perspectiva de *concerns*[2] específicos deste sistema (ISO, 2011). Uma única visão da arquitetura estabelece convenções para a construção, interpretação e uso da visão arquitetural com o intuito de conceber *concerns* específicos do sistema (ISO, 2011).

As visões arquiteturais podem ser representadas por "4 + 1" visões. Estas visões modelam a arquitetura de *software-intensive systems*, com base no desenvolvimento de pontos de vista simultâneos e de diferentes perspectivas. Utilizar as múltiplas visões permite abordar separadamente os *concerns* dos *stakeholders* de um sistema, tais como usuário final, desenvolvedores, engenheiros de sistemas, gerentes de projeto. Além disso, os requisitos funcionais e não funcionais são cuidadosamente lidados separadamente. As visões "4 + 1"' são projetadas usando um processo de desenvolvimento iterativo centrado na arquitetura, baseado em cenários (KRUCHTEN, 1995).

2.2 Architecture Description Language (ADL)

Uma ADL descreve uma arquitetura de software e fornece um ou mais tipos de modelos para expressar os *concerns* de seus *stakeholders* organizados opcionalmente em visões arquiteturais. Muitas vezes uma ADL é suportada por ferramentas automatizadas para auxiliar na criação, uso e análise de seus modelos (ISO:IEC:IEEE:42010, 2011).

As ADLs foram classificadas em duas gerações (MEDVIDOVIC; DASHOFY; TAYLOR, 2007), (MALAVOLTA et al., 2013). A primeira geração abrange os anos de 1990 a 2000.

[2] No contexto da descrição arquitetural, o termo em inglês *concern* pode ser representado em português como preocupação, interesse, expectativa, necessidade, objetivo, responsabilidade, restrição, dependência, atributo de qualidade, risco, entre outros.

Essa geração teve como principal propósito conceber uma linguagem ideal para suporte dos componentes, especificação de conectores e interconexão geral dos módulos do sistema. Pode-se citar UniCon, C2 e Darwin como exemplos de linguagens de primeira geração (PERRY; WOLF, 1992), (GARLAN; SHAW, 1994), (MALAVOLTA et al., 2013). A segunda geração compreende os anos de 2000 até o momento. Essa geração surgiu por causa das novas exigências em lidar com a crescente complexidade dos softwares modernos (DASHOFY; HOEK; TAYLOR, 2001), (PINTO; FUENTES; TROYA, 2003), (MALAVOLTA et al., 2013). As ADLs da segunda geração fornecem suporte para modelagem de gerenciamento de configuração, distribuição e linhas de produtos (MALAVOLTA et al., 2013). SOFA, PRISMA, AADL e COSA são exemplos de ADLs de segunda geração (OZKAYA; KLOUKINAS, 2013).

Todas as ADLs compartilham objetivos de abstrair os detalhes de implementação e capturar a arquitetura ao nível elevado de um software, porém existem diferenças em termos do que elas oferecem (BASHROUSH et al., 2005). Por este motivo, as ADLs são comparadas por meio de critérios de descrição arquitetural na Seção 4.2 e brevemente apresentadas na Seção 4.3.

2.3 SysML

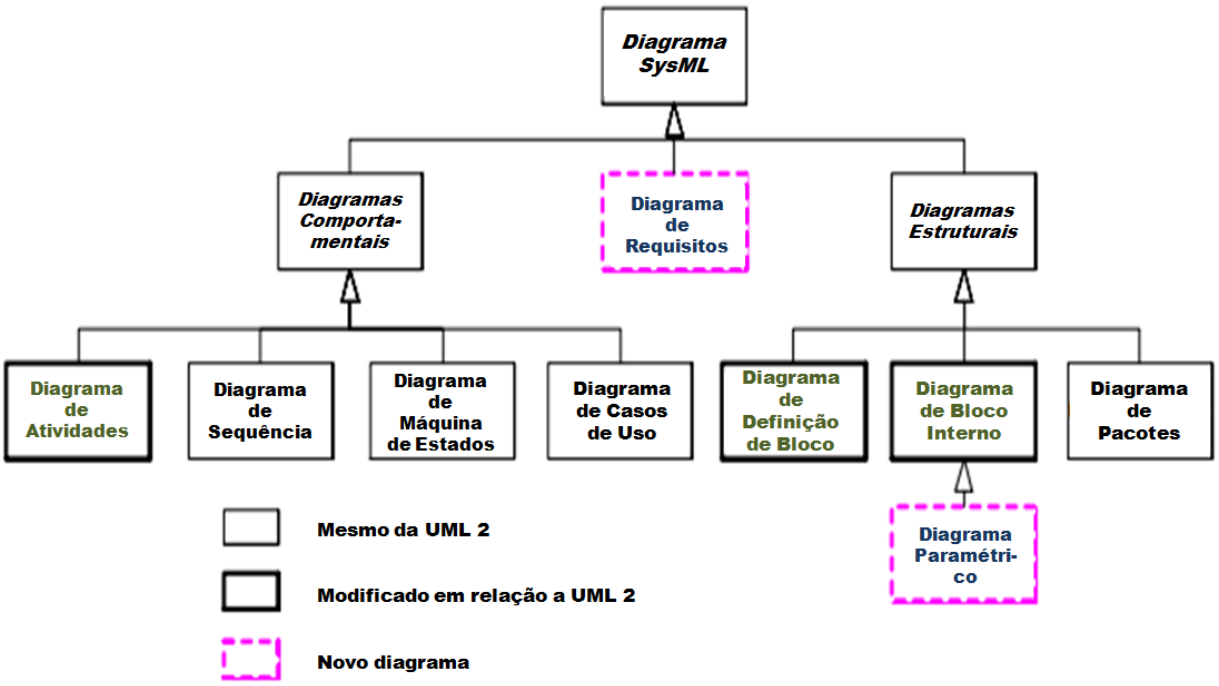

Figura 2.1 – Taxonomia dos Diagramas da SysML. Adaptada de OMG (2015).

SysML foi desenvolvida pelo *Object Management Group* (OMG) e *International Council on Systems Engineering* (INCOSE) com o objetivo de desenvolver uma linguagem unificada para modelagem de propósito geral para aplicações em engenharia de sistemas. A SysML suporta especificação, análise, projeto, verificação e validação de um grande conjunto de sistemas complexos. A linguagem é um *profile* da UML 2.0 aplicada a sistemas que incluem hardware,

software, informação, processos, pessoas e procedimentos (OMG, 2015). Atualmente, a SysML encontra-se na versão 1.4, versão esta lançada em setembro de 2015 (OMG, 2015).

Conforme a Fig. 2.1, os diagramas de Sequência, de Máquina de Estados, de Casos de Uso e de Pacotes não foram alterados em relação à UML 2.0, exceto por terem foco não apenas em software mas também em sistemas. No entanto, os diagramas de Atividade, de Definição de Bloco e de Bloco Interno foram modificados na SysML e os diagramas de Requisitos e Paramétrico são novos. Os diagramas da SysML são brevemente descritos a seguir.

Diagrama de Requisitos

O diagrama de Requisitos fornece uma modelagem para requisitos baseados em texto e Tabela (OMG, 2015). Este diagrama modela e descreve os requisitos do sistema, como os requisitos não funcionais, além de mostrar os diversos tipos de relacionamentos entre diferentes requisitos (SOARES; VRANCKEN, 2008), entre outros elementos do modelo (tais como pacotes e casos de teste) que satisfazem ou verificam os requisitos, além dos relacionamentos dos requisitos com outros diagramas (OMG, 2015).

Diagrama de Atividades

O diagrama de Atividades é usado para descrever o fluxo de controle e fluxo de entradas e saídas entre as ações de um sistema. O diagrama de Atividades é um dos diagramas da SysML que representa o comportamento e a dinâmica do sistema. Neste diagrama é apresentado o que será executado, qual a sequência, quais os dados que serão requisitados e apresentados. Por meio de uma biblioteca de dados, SysML estende o controle das ações da UML para suportar a desativação de ações que já estão sendo executadas. Assim, um operador de controle pode representar uma operação lógica complexa que transforma suas entradas para produzir uma saída que controla outras ações (OMG, 2015).

Diagrama de Sequência

O diagrama de Sequência descreve o fluxo de controle entre atores e sistemas ou entre partes de um sistema. Esse diagrama representa o envio e recebimento de mensagens entre as entidades que interagem ao longo do tempo em que uma operação é executada (OMG, 2015).

Diagrama de Máquina de Estados

O diagrama de Máquina de Estados tem o propósito de representar o comportamento do sistema mostrando os estados de um objeto e as transições entre esses estados que ocorrem em resposta a eventos. Ou seja, o comportamento do sistema é representado como um histórico de transições de estados de um objeto (OMG, 2015).

Diagrama de Casos de Uso

O diagrama de Casos de Uso descreve como o sistema é utilizado por seus atores para alcançar um objetivo. Um caso de uso pode ser visto como uma funcionalidade do sistema que é realizada por meio da interação entre o sistema e seus atores. Esse diagrama inclui casos de uso, atores e a comunicação associada entre eles. Atores representam papéis que são externos ao sistema que podem corresponder a usuários, sistemas ou outras entidades (OMG, 2015).

Diagrama de Definição de Bloco

Os Blocos da SysML são unidades modulares usadas para a descrição de sistemas. A noção de blocos em SysML permite melhor expressão da semântica de engenharia de sistemas quando comparada à UML e, particularmente, reduz o viés de UML para o software (BEHJATI et al., 2011). Os blocos descrevem tipos específicos de componentes, conexões entre componentes e a forma como esses elementos são combinados para definir o sistema completo, eles podem ser usados em todas as fases de especificação e design do sistema. Os blocos e seus relacionamentos são descritos pelo Diagrama de Definição de Bloco da SysML e sua estrutura interna pelo Diagrama de Bloco Interno da SysML (OMG, 2015).

O Diagrama de Definição de Bloco (BDD) é baseado no diagrama de Classes da UML, porém com restrições e extensões definidas para SysML. Esse diagrama define a estrutura do sistema por blocos, as operações que podem ser executadas pelo bloco são descritas pelo diagrama, assim como as relações entre os blocos, como associações, generalizações e dependências (HAUSE et al., 2006), (OMG, 2015).

Diagrama de Bloco Interno

A definição de um bloco na SysML também pode ser detalhada especificando suas propriedades, partes, portas (que especifica os pontos de interação) e seus conectores (que especifica as conexões entre suas partes e portas) (BEHJATI et al., 2011).

O Diagrama de Bloco Interno (IBD) descreve a estrutura interna de um componente em termos de propriedades e conectores do sistema. Por exemplo, a interação e comunicação entre as partes pode ser representada no Diagrama de Bloco Interno (OMG, 2015).

Diagrama de Pacotes

O diagrama de Pacotes é usado para organizar o software em pacotes e pontos de vista. Nos pacotes são agrupados os elementos lógicos que apresentam dependências entre eles. Os pacotes também podem ser utilizados para ilustrar a arquitetura de software, como as camadas, subsistemas, pacotes, entre outros (OMG, 2015).

Diagrama Paramétrico

O diagrama Paramétrico descreve restrições entre as propriedades do sistema associadas aos blocos, como propriedades de desempenho e confiabilidade para dar suporte a análise do sistema (HAUSE et al., 2006), (OMG, 2015).

2.4 Sistemas Embarcados

Sistemas eletrônicos embarcados são sistemas computacionais de uso específico, com recursos computacionais como memória e poder de processamento projetados restritamente para um único propósito (MARWEDEL; GOOSSENS, 2013). Quando estes sistemas não dependem apenas de seu comportamento funcional, mas também de seu comportamento temporal, eles são classificados como sistemas embarcados de tempo real (MARWEDEL; GOOSSENS, 2013), (MARQUES; SIEGERT; BRISOLARA, 2014). Sistemas de tempo real respondem a eventos externos em tempo hábil e o tempo de resposta é garantido (LI; YAO, 2003).

Muitos desafios e dificuldades ocorrem ao desenvolver sistemas de tempo real devido às suas características específicas, tais como mobilidade, segurança ou restrições de tempo real (KOOPMAN, 2010), (MARQUES; SIEGERT; BRISOLARA, 2014). Estes sistemas combinam hardware e software, e interagem continuamente com o ambiente a sua volta por meio de sensores e atuadores.

Na Figura 2.2 é apresentado um sistema embarcado como um simples bloco. Há um conjunto de entradas do sistema controlado. O sistema embarcado executa um programa dedicado para esta aplicação, permanentemente armazenada na memória. Diferentemente do computador pessoal de propósito geral, que executa diversos programas, o sistema embarcado sempre executa apenas um programa. Baseado na informação fornecida pelas variáveis de entrada, o microcontrolador calcula determinadas saídas, que são conectadas aos atuadores e a outros dispositivos dentro do sistema. O circuito eletrônico, juntamente com os componentes eletromecânicos, são frequentemente chamados de hardware, e o programa em execução é chamado de software. Além disso, pode existir também a interação com o usuário, (por exemplo, teclado ou tela) e interação com outros subsistemas. A variável tempo afeta tudo programado no sistema embarcado, o tempo está representado como uma seta atravessando a Figura 2.2. É necessário medir o tempo para as funcionalidades do sistema ocorrerem em tempos precisamente predeterminados, tais como gerar fluxo de dados ou sinais com uma forte dependência de tempo, e respostas a eventos inesperados em tempo hábil (TOULSON; WILMSHURST, 2016).

Os sistemas são projetados e podem ser configurados com um ou mais dos seguintes critérios: hardware, software, dados, humanos, processos, procedimentos, instalações, materiais e entidades que ocorrem naturalmente (ISO, 2008). *Software-intensive systems* é qualquer sistema em que o software contribui com influências essenciais para a concepção, construção, implantação e evolução do sistema, tais como aplicações individuais, sistemas no sentido

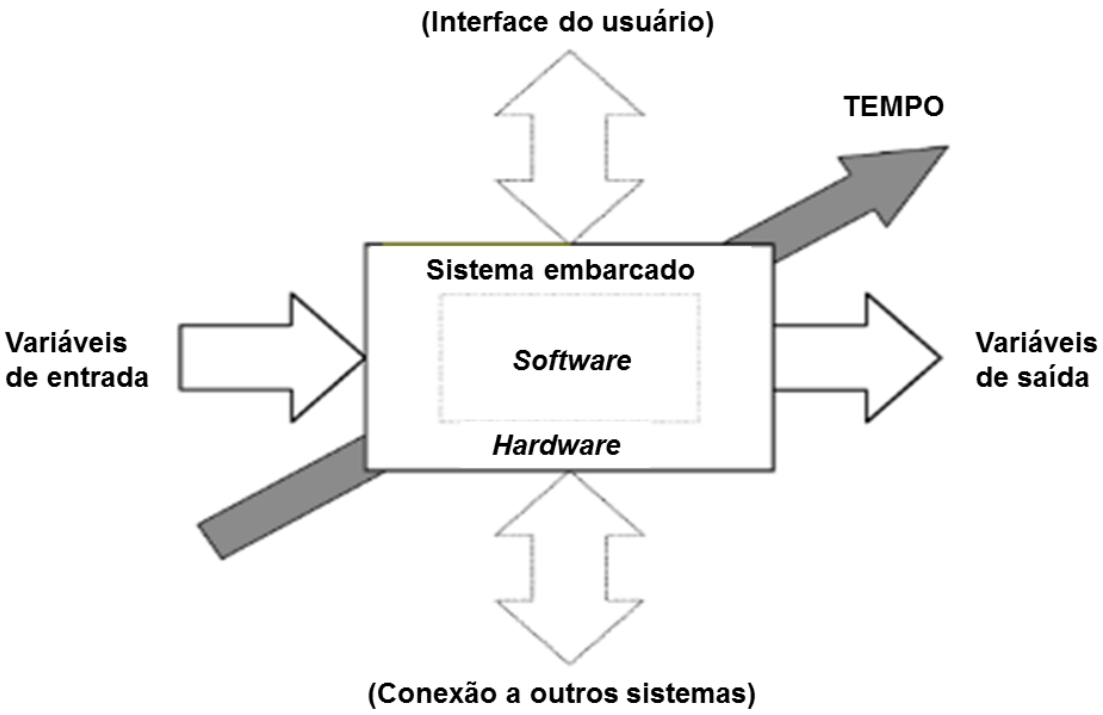

Figura 2.2 – Sistema embarcado. Adaptado de Toulson e Wilmshurst (2016).

tradicional, subsistemas, sistemas de sistemas e linhas de produtos de software (ISO, 2011), (HILLIARD, 2000).

O sistema embarcado também interage com diversas disciplinas mostradas na Figura 2.3.

Figura 2.3 – Multidisciplinaridade dos sistemas embarcados.

Engine Control Unit (ECU) é um componente físico que possui lógica necessária para controle de sensores, atuadores e as diferentes ações que representam atividades de controle de sistemas embarcados de um automóvel. Os ECUs têm a responsabilidade de coordenação de atuadores que fornecem o comportamento esperado para cada componente, como também comunicação e controle dos demais ECUs incorporados no sistema.

2.5 Sistemas Embarcados Automotivos

Sistemas embarcados estão remodelando a forma como as pessoas vivem, trabalham e divertem. Automóveis equipados com sistemas de navegação auxiliam a encontrar destinos e rotas para pessoas de forma segura e eficiente. Ouvir músicas favoritas, abrir e fechar janelas do carro, acender faróis automaticamente são exemplos comuns. A maioria dos veículos tem chips de computador que executam tarefas automáticas, que tornam os veículos mais fáceis de dirigir (LI; YAO, 2003), (SPAAN et al., 2016). Isso significa que a maioria dos carros novos está executando dezenas de milhões de linhas de código de software, controlando tudo, desde os freios até o volume do rádio de um veículo (CHARETTE, 2010).

A arquitetura eletrônica automotiva de um veículo moderno contém entre 70 a 100 ECUs e um veículo comum contém entre 30 a 50 ECUs (CHARETTE, 2010). Para comunicações, normalmente um veículo comum tem dois ou três Barramentos (BUS) de *Controller Area Network* (CAN), com taxas de 25 a 500 *Kbps*, dois ou três barramentos de rede de baixa velocidade que fazem a interconexão local e, opcionalmente, alguns links de alta velocidade para troca de informações. As aplicações de segurança são planejadas para serem executadas, normalmente a taxas entre 20 a 100 ms. Estas aplicações apresentam desafios devido a alta distribuição, complexidade e interoperabilidade (SANGIOVANNI-VINCENTELLI; NATALE, 2007).

Os sistemas embarcados automotivos são sistemas de tempo real que podem ser divididos entre domínios funcionais “centrados nos veículos”, tais como controle de trem de força (motor e sistema de transmissão), controle de chassi e sistemas de segurança ativos ou passivos, e domínios funcionais “centrados nos passageiros” que podem ser identificados nos sistemas de multimídia/telemática, corpo/conforto e interface homem-máquina (ZURAWSKI, 2009).

O domínio funcional “centrado no passageiro” identificado no corpo/conforto contém funções incorporadas em um veículo que estão relacionadas aos sistemas de limpadores de vidro, de faróis, de portas, de janelas, de poltronas, de *airbag* e de retrovisores. Estas funções são controladas cada vez mais por sistemas baseados em software nos últimos anos.

Geralmente, estes sistemas não estão sujeitos a restrições rigorosas de desempenho. Do ponto de vista da segurança, eles não representam uma parte tão crítica do sistema. No entanto, existem certas funções que têm de respeitar restrições de tempo real, como um exemplo, o sistema de *airbag* cujo objetivo é a segurança dos passageiros (ZURAWSKI, 2009).

No sistema automotivo, a descrição da arquitetura normalmente é realizada antes do desenvolvimento e integração dos subsistemas. Neste processo, modelos da arquitetura de software e possíveis soluções para a arquitetura física devem ser definidos e combinados para avaliar a qualidade em relação ao desempenho, confiabilidade, custo e restrições. Dado o alto custo de pesquisa, treinamento e possivelmente aquisição de licenças para o projeto em nível de sistema, é desejável usar um conjunto coerente de modelos, métodos e ferramentas durante a vida útil de um produto ou plataforma. Isso se estende do estágio de análise de arquitetura ao design do sistema e inclui o desenvolvimento baseado em modelos, geração automática de código e os estágios finais de integração, teste e validação (SANGIOVANNI-VINCENTELLI; NATALE, 2007).

3 Especificação de Requisitos de Sistemas Embarcados de Tempo Real

Neste capítulo serão apresentados os exemplos para a construção do projeto arquitetural de sistema embarcado automotivo de tempo real. Primeiramente, é explanada brevemente a Norma ISO/IEC/IEEE 29148:2011 na Seção 3.1. Posteriormente, na Seção 3.2 o sistema de controle de *airbag* é explicado e os requisitos que compõem o sistema são apresentados. Em seguida, na Seção 3.3 é mostrada a mesma análise para o sistema de controle de faróis.

3.1 Especificação de Requisitos usando a Norma ISO/IEC/IEEE 29148:2011

A Norma ISO 29148:2011 reúne práticas recomendadas e guias com o intuito de prover um documento padrão para os arquitetos de software especificarem os requisitos dos sistemas e projetos de software ao longo de todo o ciclo de vida.

Segundo a ISO (2011), os requisitos devem ser cuidadosamente especificados e devem capturar as reais necessidades dos *stakeholders* de forma acurada.

Um requisito bem desenvolvido deve ser verificado, ser satisfeito por um sistema para resolver um problema ou atingir um objetivo dos *stakeholders*. O requisito também deve ser qualificado por condições mensuráveis e limitado por restrições. Além disso, o requisito define o desempenho do sistema quando utilizado por um *stakeholder* específico ou a capacidade relacionada ao sistema.

Uma definição de requisito é que este expressa ou traduz uma necessidade associada às suas restrições e condições. O requisito expresso em linguagem natural não deve ser ambíguo.

A descrição natural deve compreender um sujeito, um verbo e um complemento, por exemplo, o sujeito pode ser “sistema” ou “software”, o verbo e o complemento representam o que deve ser feito, como “desligar faróis frontais em exatos 3 minutos”. É importante concordar com antecedência sobre as palavras-chave e os termos específicos que sinalizam a presença de um requisito. A abordagem comum é utilizar a palavra “deve”, pois um requisito é obrigatório, a palavra “pode”, em vez de “deve” pode causar ambiguidade. Consequentemente, é recomendado evitar voz passiva, uso de verbos no futuro, futuro do pretérito e pretérito. É recomendado também usar descrições positivas, em vez de negativas, como o “não deve”. Todos os termos específicos para engenharia de requisitos devem ser formalmente definidos e aplicados consistentemente em todos os requisitos do sistema. Na Figura 3.1 são apresentados exemplos de sintaxe de requisitos.

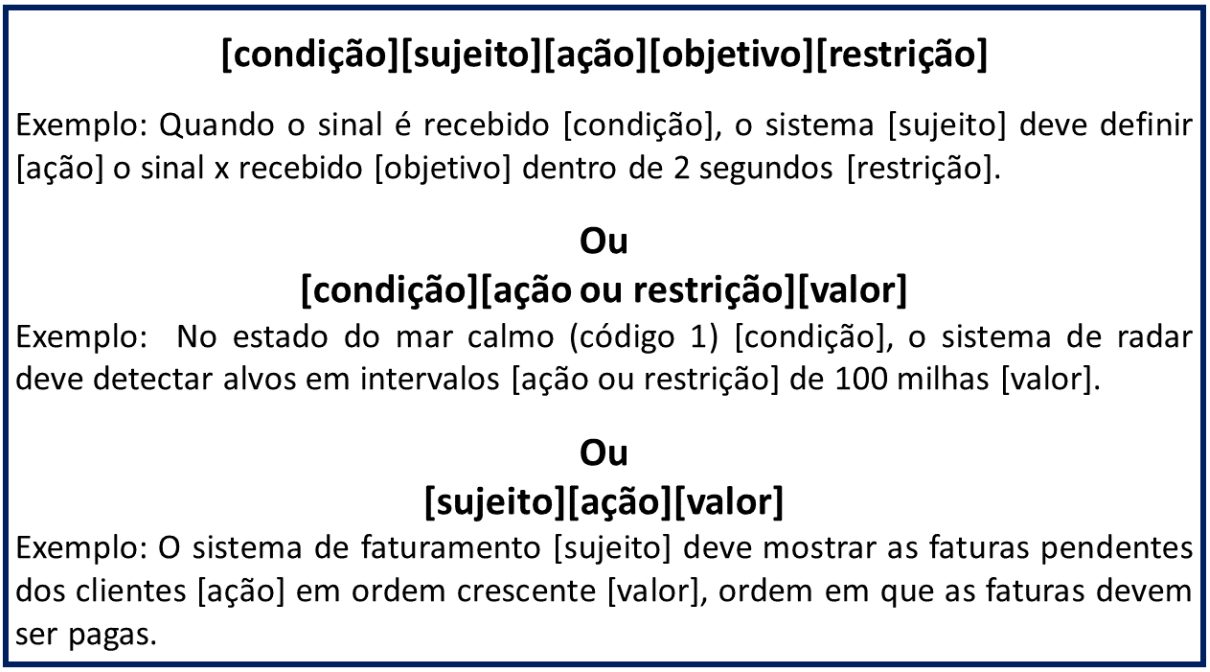

Figura 3.1 – Exemplos de sintaxe de um requisito. Adaptada de ISO (2011).

Na Figura 3.1, a condição (representada por **[condição]**) é um atributo qualitativo ou quantitativo mensurável, que é estipulado para um requisito, a condição pode ser validada e verificada.

Na Figura 3.1, a restrição (representada por **[restrição]**) restringe a solução de um projeto ou a implementação do processo de engenharia de sistemas. Por exemplo, leis aplicadas a determinado país, duração ou orçamento disponíveis, plataforma tecnológica, capacidades e limitações dos usuários. As restrições podem ser aplicadas em todos os requisitos, podem ser especificadas em um relacionamento com um requisito específico ou conjunto de requisitos, ou podem ser identificados como requisitos independentes.

3.1.1 Especificação de Requisitos de Software

A *Software Requirements Specification* (SRS) (Ver o exemplo de esquema na Figura 3.2) é uma especificação de requisitos para um determinado produto de software, programa ou conjunto de programas que executa determinadas funções em um ambiente específico. A SRS pode ser escrita por um ou mais representantes do fornecedor, um ou mais representantes do cliente, ou por ambos.

É importante considerar que a SRS engloba o plano do projeto. O software pode conter essencialmente toda a funcionalidade do projeto ou pode ser parte de um sistema maior. No último caso, tipicamente haverá uma especificação de requisitos que indicará as interfaces entre o sistema e a sua porção de software. A SRS indica a precedência e criticidade dos requisitos e também define todos os recursos necessários do produto de software especificado ao qual

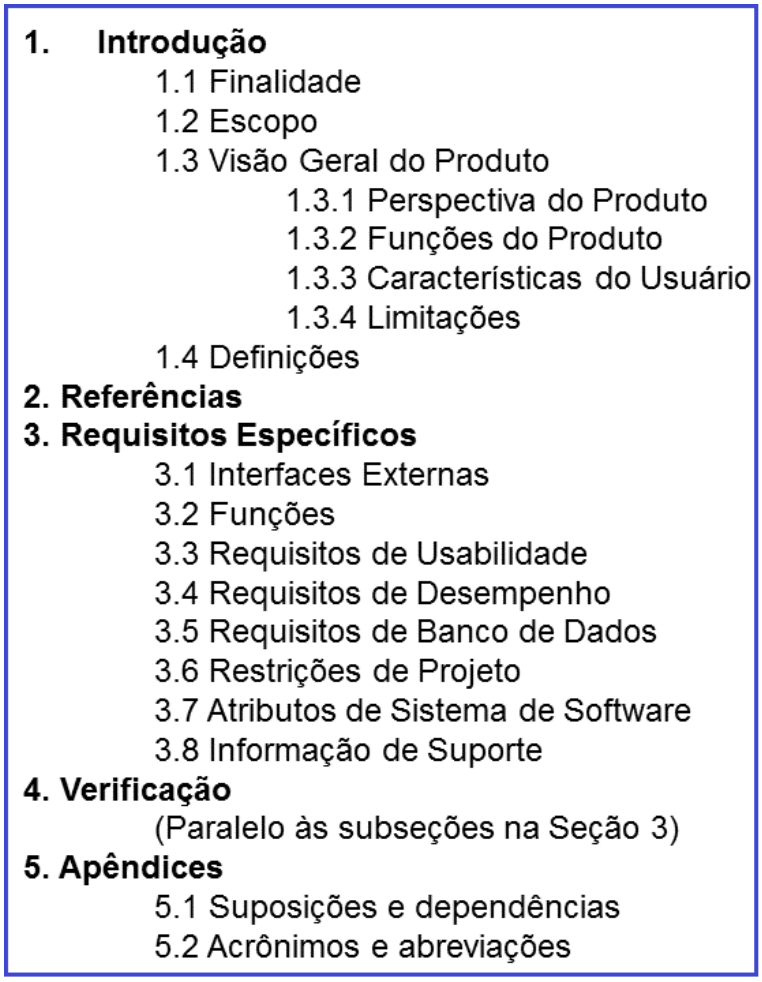

Figura 3.2 – Exemplo de esquema SRS. Adaptada de ISO (2011).

ele se aplica, bem como documenta as condições e restrições sob as quais o software deve ser executado e as abordagens de verificação previstas para os requisitos.

A organização dos itens de informação no documento SRS, tais como a estrutura, ordem e seção, pode ser selecionada de acordo com as políticas de documentação do projeto que está sendo construído.

3.2 Sistema de Controle de *Airbag*

Nesta Seção é apresentado o documento de especificação de requisitos de software para o sistema de controle de farol, de acordo com a Norma ISO/IEC/IEEE 29148:2011. Os requisitos de software apresentados nesta Seção primeiramente foram elicitados por meio de leitura e pesquisa dos manuais de automóveis[3], *web-sites* (SUPER-INTERESSANTE, 2016), (BRAGA, 2016) e do livro (ZURAWSKI, 2009). Após essa etapa os requisitos foram identificados e classificados para o sistema de controle de *airbag*.

[3] Fiat. Manual do seu carro Fiat. Disponível em: <http://www.fiat.com.br/manual-do-seu-fiat.html>, acesso em 20 de abril de 2016.

1. **Introdução**

   O *airbag* é um item de segurança capaz de amenizar o impacto de uma possível colisão e proteger a vida dos passageiros. Aliado ao cinto de segurança, o *airbag* protege o motorista e eventual passageiro contra os efeitos de um impacto frontal no veículo.

   O ativamento do *airbag* ocorre somente se o impacto ocorrer dentro de um ângulo máximo de 30 graus em relação ao movimento frontal. Os sensores que determinam o instante do disparo de um *airbag* são incorporados à unidade de comando do sistema. O sistema não é ativado em caso de impactos laterais ou traseiros ou ainda em caso de capotamento.

   a) **Finalidade**

      O sistema de controle de *airbag* deve calcular, analisar e verificar se ocorre uma colisão no veículo, caso ocorra, o sistema deve estar preparado para ativar o *airbag*, de acordo com as restrições previstas nos requisitos.

   b) **Escopo**

      O sistema de *airbag* é nomeado por *AirbagSystem*.

      O *AirbagSystem* comunica com outras partes do veículo para obter todas as informações necessárias para a segurança dos passageiros.

   c) **Visão Geral do Produto**

      i. **Perspectiva do Produto**
      A comunicação é realizada com controladores de velocidade, painel de controle, peso, comunicação, além dos atuadores e sensores do sistema de *airbag*. Toda a comunicação é rigorosamente calculada pelo tempo de relógio, ou seja, a restrição de tempo neste sistema é um fator importante.

      ii. **Funções do Produto**

      - Obter informações sobre colisão
      - Reconhecer colisão
      - Calcular peso de passageiros e carro
      - Calcular movimento
      - Calcular ângulo
      - Estar sempre ativo quando o carro está ligado
      - Analisar ativamento do *airbag*
      - Ativar *airbag*

      iii. **Características do Usuário**
      O *airbag* faz parte do sistema veicular, mas não há interação com quaisquer usuários. O sistema atua como segurança para os passageiros e todas as funcionalidades são realizadas automaticamente. Caso ocorra algum problema com qualquer parte do *airbag*, o motorista é comunicado pelo painel de controle.

Dessa forma, deve-se levar o carro para um especialista averiguar o defeito ocorrido.

2. **Requisitos Específicos**

   a) **Interfaces externas**

   - Sistema de controle de velocidade
   - Painel
   - Sistema de controle de peso
   - Sistema de controle de comunicação
   - Atuadores
   - Sensores

   b) **Requisitos Funcionais**

   Neste exemplo, de acordo com as características listadas anteriormente, e os resultados da análise detalhada de domínio, os requisitos para a modelagem de tal sistema são:

   RF1: O sistema de controle de *airbag* deve receber o sinal do sensor de velocidade do carro a cada 15 ms.

   RF2: O sistema de controle de *airbag* deve avaliar o sinal do sensor de velocidade do carro a cada 15 ms.

   RF3: O sistema de controle de *airbag* deve reconhecer em no máximo 5 ms uma variação brusca de velocidade de pelo menos 20 km/h.

   RF4: Quando o carro é ligado, o sistema de controle de *airbag* deve calcular o peso do carro somado ao peso dos ocupantes em no máximo 5 s.

   RF5: Caso o requisito RF3 seja satisfeito, o sistema de controle de *airbag* deve calcular o ângulo do impacto da colisão em exatos 5 ms.

   RF6: O sistema de controle de *airbag* deve reconhecer o ângulo do impacto da colisão em no máximo 5 ms.

   RF7: O sistema de controle de *airbag* somente deve ativar o *airbag* se o ângulo do impacto for menor que 30 graus.

   RF8: O sistema de controle de *airbag* somente deve ativar o *airbag* se o impacto da colisão for em movimento frontal.

   RF9: O sistema de controle de *airbag* deve ativar o *airbag* em no máximo 15 ms após impacto da colisão.

   RF10: O sistema de controle de *airbag* deve agendar solicitações simultâneas a cada 15 ms.

   RF11: O sistema de controle de *airbag* deve receber o sinal do sensor de *airbag* a cada 15 ms.

RF12: O sistema de controle de *airbag* deve reconhecer o estado do atuador de *airbag* a cada 5 ms.

RF13: O sistema de controle de *airbag* deve avaliar o funcionamento de todos os componentes do *airbag* a cada 15 ms.

RF14: O sistema de controle de *airbag* deve notificar ao usuário (com uma luz no painel) o mau funcionamento de algum componente do *airbag* em no máximo 50 ms.

RF15: O sistema de controle de *airbag* deve executar as solicitações contidas no RF10 a cada 10 ms.

c) **Requisitos não funcionais**

RNF01: O sistema de controle de *airbag* deve estar apto a detectar e evitar causas de colisão.

RNF02: O carro deve possuir sistema de *airbag* confiável, testes devem garantir menor taxa (0,10%) a falhas de software.

RNF03: O sistema de controle de *airbag* deve usar protocolos seguros na transmissão de dados.

RNF04: Quando o carro está ligado, o sistema de controle de *airbag* deve estar disponível 99,9999 % do tempo.

RNF05: O sistema de controle de *airbag* deve atender as normas legais da ABNT NBR 15300-1, 15300-2, 15300-3 e ISO 3833.

RNF06: O sistema de controle de *airbag* deve ser íntegro, todos os sistemas que se comunicam com o sistema de *airbag* devem ser identificados, monitorados e impedidos de alteração de dados, a não ser que sejam autorizados.

d) **Restrições de Projeto**

O sistema deve ser implementado com protocolos de segurança e restrições na comunicação entre os sistemas. O sistema deve também incorporar as normas legais definidas no requisito não funcional RNF05.

e) **Atributos de Sistema de Software**

- Confiabilidade: Estabelece a confiabilidade requerida do sistema de software em tempo real.
- Segurança: Restringe comunicações entre as partes do sistema de software.
- Segurança: Assegura a privacidade dos dados.
- Disponibilidade: O sistema deve estar disponível sempre que o carro estiver ligado.

f) **Informação de Suporte**

Quando ocorre uma colisão frontal em que existe a possibilidade de ativação de *airbag* é a desaceleração brusca do automóvel que ativa o dispositivo de *airbag*. O

sensor identifica quando a velocidade varia pelo menos 20 quilômetros por hora em um curto espaço de tempo.

A ECU envia um sinal elétrico para o ignitor, responsável por inflar o *airbag*. Dentro do ignitor, as substâncias como os nitratos de amônia e guanidina reagem e explodem instantaneamente. A reação química gera nitrogênio suficiente para encher a bolsa (que pode ter entre 60 e 90 litros) em apenas 30 ms. O *airbag* começa a esvaziar no impacto com o corpo.

A bolsa contém um gás gerador químico em estado sólido. Esses gases ficam armazenados numa câmara de metal dentro do módulo do *Airbag*. As bolsas são lacradas. Quando a unidade de comando recebe o sinal de impacto do veículo, uma corrente elétrica é aplicada à bolsa provocando a ignição do gás gerador. Esse gás queima rapidamente na câmara de metal fazendo com que seja produzida uma certa quantidade de nitrogênio e dióxido de carbono que se expandem enchendo a bolsa.

3.3 Sistema de Controle de Faróis

Nesta Seção é apresentado o documento de especificação de requisitos de software para o sistema de controle de farol, de acordo com a Norma ISO/IEC/IEEE 29148:2011.Os requisitos de software apresentados nesta Seção primeiramente foram elicitados por meio de leitura e pesquisa dos manuais de automóveis, *web-site* (CARROS-ONLINE, 2016) e do livro (ZURAWSKI, 2009). Após essa etapa os requisitos foram identificados e classificados para o sistema de controle de faróis automáticos.

1. **Introdução**

   No sistema de controle de farol, os faróis automáticos são ativados por meio de um sensor fotoelétrico. O sensor é ativado ou desativado por condições de iluminação, podendo ser escuro ou claro, respectivamente. Dessa forma, o controle automático de faróis acende as luzes sempre que o sensor sentir o ambiente escuro. Por exemplo, o sistema ativará os faróis quando o carro entrar em um túnel ou quando o ambiente apresentar nuvens pesadas.

   a) **Finalidade**

      O sistema de controle de faróis está sendo projetado para que o acendimento dos faróis de um veículo seja feito de forma automática, segura e adequada.

   b) **Escopo**

      O sistema de controle de farol é nomeado por LightSystem.

      Além de LightSystem oferecer o acendimento automático dos faróis, ele permite a iluminação do caminho da frente do veículo que permanece ligada por exatos três minutos após o desligamento do motor do carro. Essa funcionalidade serve para que

quando os clientes do sistema cheguem em casa a noite permaneçam com o caminho iluminado. Além disso, o sistema pode ser ativado ou desativado pelo motorista em qualquer momento.

c) **Visão Geral do Produto**

i. **Perspectiva do Produto**
O LightSystem interage com sensor, atuador e com outros sistemas tais como controladores de motor, comunicação e painel do carro.

ii. **Funções do Produto**

- Ativar os faróis do carro automaticamente
- Desativar os faróis do carro automaticamente
- lightSystem pode ser ativado e desativado a qualquer momento pelo motorista
- lightSystem permanece ativado por 3 minutos após o carro ser desligado

iii. **Características do Usuário**
Os usuários do sistema são motoristas.

2. **Requisitos Específicos**

a) **Interfaces externas**

- Sistema de controle de painel
- Sistema de controle de motor
- Sistema de controle de comunicação
- Atuador
- Sensor

b) **Requisitos Funcionais**

Neste exemplo, de acordo com as características listadas anteriormente, e os resultados da análise detalhada de domínio, os requisitos funcionais para a modelagem de tal sistema são:

RF1: O sistema de controle de faróis deve desligar faróis frontais em exatos 3 minutos após o motor do carro ser desligado.

RF2: O sistema de controle de faróis deve agendar solicitações simultâneas a cada 50 ms.

RF3: O sistema de controle de faróis deve avaliar o funcionamento de todos os componentes de faróis a cada 50ms.

RF4: O sistema de controle de faróis deve reconhecer o status do sensor dos faróis frontais a cada 50ms.

RF5: O sistema de controle de faróis deve reconhecer quando o sensor detectar o ambiente escuro em no máximo 50 ms.

RF6: Caso RF5 seja satisfeito, o sistema de controle de faróis deve ligar os faróis em no máximo 50ms.

RF7: O sistema de controle de faróis deve reconhecer quando o sensor detectar o ambiente claro em no máximo 50 ms.

RF8: Caso RF7 seja satisfeito, o sistema de controle de faróis deve desligar os faróis em no máximo 50ms.

RF9: O sistema de controle de faróis deve receber sinal do sensor de faróis a cada 50ms.

RF10: O sistema de controle de faróis deve reconhecer o estado do atuador de faróis a cada 50ms.

RF11: O sistema de controle de faróis deve notificar ao usuário (com uma luz no painel) o mau funcionamento de algum componente de faróis em no máximo 50ms.

RF12: Caso RF5 seja satisfeito, o sistema de controle de faróis deve modificar intensidade (nível 1 - farol de lanterna, 2 - farol baixo e 3 - farol alto) dos faróis em no máximo 50ms, quando houver variação de iluminação externa.

RF13: O sistema de controle de faróis deve permitir que o motorista possa desabilitar o sistema de faróis automáticos.

RF14: O sistema de controle de faróis deve permitir que o motorista possa habilitar o sistema de faróis automáticos.

RF15: O sistema de controle de faróis deve permitir que o motorista possa ligar os faróis manualmente.

RF16: O sistema de controle de faróis deve permitir que o motorista possa modificar a intensidade (nível 1 - farol de lanterna, 2 - farol baixo e 3 - farol alto) dos faróis manualmente.

RF17: O sistema de controle de faróis deve permitir que o motorista possa desligar os faróis manualmente.

RF18: Quando o motorista usa o sistema de controle de faróis manualmente, o sistema deve notificar ao usuário (por meio de som) caso o motorista deixe o farol aceso, após motor ser desligado.

RF19: O sistema de controle de faróis deve executar as solicitações contidas no RF2 a cada 50 ms.

c) **Requisitos não funcionais**

RNF01: O carro deve possuir sistema de faróis confiável, testes devem garantir no mínimo 99% de taxa relacionadas as falhas de software.

RNF02: Quando o carro está ligado, o sistema de controle de faróis deve estar disponível 99 % do tempo.

d) **Restrições de Projeto**

O lightSystem apenas pode ser ativado ou desativado pelo motorista. Os outros sistemas que interagem com ele possuem restrições de comunicação.

e) **Atributos de Sistema de Software**

- Segurança: Restringe comunicações entre as partes do sistema de software.
- Disponibilidade: O sistema deve estar disponível sempre que o carro estiver ligado.

f) **Informação de Suporte**

O sensor fotoelétrico deve capturar as condições ambientais adequadas para a ativação automática dos faróis, tais como chuva, túnel, noite, nuvens pesadas.

A temperatura de cor determina a aparência, a cor da luz. A emissão da luz é classificada de acordo com o seu espectro. O espectro de emissão é determinado pela distribuição de sua energia segundo diferentes comprimentos de ondas, medido em nanômetros (nm). Dessa forma, cada cor tem um valor em nanômetros. Um nanômetro equivale a um milionésimo de milímetro.

4 Uma Análise Comparativa de Linguagens de Descrição de Arquitetura de Software

Este capítulo é baseado no artigo *"A Comparative Analysis of Software Architecture Description Languages"* submetido à *International Conference on Computational Science and Its Applications* (ICCSA).

Muitas linguagens de descrição de arquitetura (ADLs) foram propostas nos últimos anos, mas isso não resultou em sua ampla utilização real na indústria de software (PANDEY, 2010), (MALAVOLTA et al., 2013), (OZKAYA; KLOUKINAS, 2013). Na prática, os arquitetos de software mais comumente usam UML como uma ADL em projetos (MALAVOLTA et al., 2013). Apesar disso, a UML não foi criada para a descrição da arquitetura, uma vez que a UML foi proposta como uma linguagem de modelagem a ser aplicada em atividades de modelagem de software orientado a objetos.

Foram considerados quatro principais objetivos para a pesquisa da identificação dos critérios e comparação das ADLs. São eles (1) especificar as principais características de ADLs, (2) identificar a evolução de ADLs, (3) identificar as vantagens de uma determinada ADL em relação as outras para garantir realmente a descrição de uma arquitetura e (4) mostrar que as especificações das arquiteturas de software podem elevar o nível de abstração de software e expandir o entendimento das partes interessadas por meio das visões arquiteturais.

A comparação das ADLs foi realizada sob a consulta de artigos científicos nas bibliotecas digitais ACM, IEEExplore, ScienceDirect e Scopus nos últimos cinco anos em língua inglesa. O estudo (MALAVOLTA et al., 2013) apresenta uma tabela contendo 20 características que uma ADL pode possuir. Os autores encontraram estas características mediante estudo bibliográfico, experiência profissional, questionários e entrevistas com profissionais que desenvolveram ADLs.

Das 20 características encontradas em Malavolta et al. (2013), 11 foram consideradas como critérios para análise comparativa, 5 foram acrescentadas mediante leitura dos trabalhos (KRUCHTEN, 2008), (PANDEY, 2010) e (OZKAYA; KLOUKINAS, 2013), e normas (SINGH, 1996), (ISO:IEC:IEEE:42010, 2011), totalizando 16 características que foram especificadas como critérios de análise comparativa entre as ADLs neste trabalho. Todas as características (Ver Tabela 4.1) foram base para a comparação entre as ADLs AADL, ABC/ADL, C2, COSA, Darwin, PRISMA, RAPIDE, SOFA, UniCon e Wright.

Tabela 4.1 – Critérios de análise comparativa entre as ADLs. Adaptado de Malavolta et al. (2013), Kruchten (2008), Pandey (2010) Ozkaya e Kloukinas (2013), Singh (1996), ISO:IEC:IEEE:42010 (2011).

Critérios
Origem
Ano de 1 publicação
Geração
Tipo de domínio (geral ou específico)
Suporte a componentes
Suporte a conectores
Suporte a múltiplas visões arquiteturais
Suporte a padrões ou estilos arquiteturais
Suporte a requisitos não funcionais
Suporte a UML
Sintaxe textual
Sintaxe gráfica
Semântica formal
Geração automática de código fonte
Ambiente de desenvolvimento
Suporte a simulação (Execução do modelo)

4.1 Critérios para a Análise Comparativa das Linguagens de Descrição de Arquitetura de Software

Nesta Seção, cada um dos critérios de análise comparativa entre as ADLs é discutido a seguir.

A origem, o ano da 1ª publicação e a geração da ADL possibilitam uma análise comparativa mais precisa entre as ADLs. A origem indica o local de criação da ADL. O ano da 1ª publicação de artigos, possivelmente refere-se ao ano da criação da ADL. A geração a qual a ADL se enquadra é obtida a partir do ano de 1ª publicação. Outro fator importante é descobrir para qual domínio a ADL foi criada, se para domínio específico ou geral.

Componentes são as menores unidades da arquitetura, ou seja, não são passíveis de decomposição no nível arquitetural. Dentro dos componentes são realizadas computações implementadas em uma linguagem de programação (PETTY; MCKENZIE; XU, 2002).

Conectores representam relações ou interações entre componentes. Conectores atuam como mediadores entre qualquer tipo de interação, como fluxo de dados e de controle entre os componentes (PETTY; MCKENZIE; XU, 2002).

Visões arquiteturais permitem abordar separadamente *concerns* dos *stakeholders*, dentre eles, usuários finais, clientes, especialistas de banco dados, desenvolvedores, engenheiros de

sistemas e gerentes de projeto. Cada visão é descrita por um modelo utilizando a sua própria notação particular. Pode-se citar *frameworks* que abordam as visões arquiteturais, entre eles 4+1 *view model* (KRUCHTEN, 1995), Zachman (ZACHMAN, 2009) e Norma ISO/IEC/IEEE 42010 (ISO:IEC:IEEE:42010, 2011). Os *frameworks* arquiteturais foram criados com a intenção de resolver o problema da arquitetura de software, mas não abordam os *concerns* de todos os *stakeholders* (KRUCHTEN, 1995).

Padrão arquitetural, muitas vezes chamado de estilo arquitetural, define uma família de sistemas nos termos de um padrão da estrutura organizacional (GARLAN; SHAW, 1994). Mais especificamente, um padrão arquitetural especifica o vocabulário dos componentes, conectores, restrições e como os padrões podem ser combinados. Para cada visão arquitetural, os arquitetos podem escolher um ou mais padrões arquiteturais, permitindo assim a coexistência de mais de um padrão em um único software (GARLAN; SHAW, 1994) (KRUCHTEN, 1995). Pode-se citar como exemplos, *Pipes and Filters, Layered Systems, blackboard e event-based* (GARLAN; SHAW, 1994).

Requisitos não funcionais (NFRs) como segurança, usabilidade, integridade e desempenho podem ser descritos na arquitetura. Caso sejam, os NFRs são mapeados diretamente da engenharia de requisitos ao projeto arquitetural (XU et al., 2005).

Desde a sua concepção em 1997, a UML atraiu organizações e profissionais. Atualmente, a UML é a linguagem de modelagem de fato para o desenvolvimento de software. A versão UML 2.0 fornece 13 tipos de diagramas que permitem a modelagem de visões arquiteturais diferentes e níveis de abstração (LANGE; CHAUDRON; MUSKENS, 2006).

Diversas linguagens provêm uma sintaxe textual e outras possuem adicionalmente sintaxe gráfica. A sintaxe gráfica nas descrições arquiteturais é outra maneira de melhorar a compreensão do sistema. No entanto, isto só é possível se há uma relação precisa entre a descrição gráfica e textual, de tal modo que as descrições sejam equivalentes (MEDVIDOVIC, 1996).

Um método formal é uma técnica baseada na matemática para descrever propriedades de hardware e sistemas de software. Sistemas completos podem ser especificados, desenvolvidos e verificados de forma sistemática. O método formal é usado por linguagens formais. Uma linguagem formal é formada pelo domínio sintático, que provê a notação da linguagem e pelo domínio semântico, que provê regras precisas para os elementos, regras essas que definem quais elementos satisfazem cada especificação. Uma especificação é uma sentença escrita em termos dos elementos do domínio sintático (WING, 1990). A semântica formal também estabelece precisão e evita ambiguidade (SCHOBBENS; HEYMANS; TRIGAUX, 2006).

As ferramentas automáticas de geração de código podem ser usadas para mapear descrições arquiteturais para o código fonte em uma determinada linguagem de programação. ADLs podem apoiar a consistência da arquitetura com a implementação de código, com o objetivo de manter o código gerado sincronizado com a especificação arquitetural (TERRA; VALENTE,

2009). Ao fazer manualmente pode resultar em problemas de consistência e rastreabilidade entre uma arquitetura e sua implementação (MEDVIDOVIC, 1996).

Muitas vezes, uma ADL é suportada por ferramentas para auxiliar a criação, uso e análise de seus modelos (ISO:IEC:IEEE:42010, 2011). Essas ferramentas são conhecidas por ambiente de desenvolvimento, tendo como exemplos, o ambiente de desenvolvimento OSATE para a linguagem AADL e Eclipse com o *plugin* Papyrus para dar suporte a SysML.

Existem ADLs que proporcionam análise em nível arquitetural, como a simulação automática (execução do modelo arquitetural). Por vezes os projetos são caros e mudanças tornam-se dispendiosas, portanto a simulação permite que decisões iniciais de um projeto possam ser testadas antes de serem implementadas, possibilitando prever possíveis comportamentos do projeto (CLEMENTS, 1996).

4.2 Tabela Comparativa das Linguagens de Descrição de Arquitetura de Software

A análise comparativa e o conjunto de características das ADLs são apresentados resumidamente na Tabela 4.3. O preenchimento da Tabela 4.3 considera uma bolinha preta (●) como um critério que é suportado, uma bolinha branca (○) significa que o critério não é suportado, uma bolinha parcialmente branca e preta (◐) indica que o critério é parcialmente suportado, e este símbolo (⊠) indica que o critério não foi especificado.

A Tabela 4.2 indica as publicações consultadas para a composição das informações obtidas na Tabela 4.3.

Tabela 4.2 – Referências.

ADLs		**Referências**
1.	AADL	(FEILER; GLUCH; HUDAK, 2006), (FEILER; LEWIS; VESTAL, 2006), (FRANCA et al., 2007), (VARONA-GOMEZ; VILLAR, 2009), (HILLIARD et al., 2012), (OZKAYA; KLOUKINAS, 2013)
2.	ABC/ADL	(MEI et al., 2002), (CHEN et al., 2002)
3.	C2	(MEDVIDOVIC, 1995), (MEDVIDOVIC, 1996), (MEDVIDOVIC et al., 1996), (TAYLOR et al., 1996), (MEDVIDOVIC; EGYED; ROSENBLUM, 1999), (OZKAYA; KLOUKINAS, 2013)
4.	COSA	(OUSSALAH; SMEDA; KHAMMACI, 2004), (SMEDA et al., 2009), (OZKAYA; KLOUKINAS, 2013)
5.	Darwin	(MAGEE; DULAY; KRAMER, 1993), (MAGEE; DULAY; KRAMER, 1994), (MAGEE; KRAMER, 1996), (MEDVIDOVIC, 1996), (MEDVIDOVIC; EGYED; ROSENBLUM, 1999), (MEDVIDOVIC; TAYLOR, 2000), (HILLIARD et al., 2012), (OZKAYA; KLOUKINAS, 2013)
6.	PRISMA	(PÉREZ et al., 2003), (PÉREZ et al., 2005), (PÉREZ et al., 2006), (PEREZ et al., 2006), (OZKAYA; KLOUKINAS, 2013)
7.	RAPIDE	(SANTORO, 1993), (LUCKHAM, 1996), (RAPIDE, 1997), (MEDVIDOVIC; TAYLOR, 2000), (MEI et al., 2002), (DAI; COOPER, 2005), (HILLIARD et al., 2012)
8.	SOFA	(PLÁŠIL; BÁLEK; JANEČEK, 1998), (PLASIL; VISNOVSKY, 2002), (BURES; PLASIL, 2004), (BUREŠ; HNĚTYNKA; PLÁŠI, 2006), (HNETYNKA; BURES, 2007), (LINHARES et al., 2007), (OZKAYA; KLOUKINAS, 2013), (ČERNỲ et al., 2015)
9.	UniCon	(ZELESNIK, 1996), (MEDVIDOVIC; EGYED; ROSENBLUM, 1999), (MEDVIDOVIC; TAYLOR, 2000), (MEI et al., 2002), (OZKAYA; KLOUKINAS, 2013)
10.	Wright	(ALLEN; GARLAN, 1996), (MEDVIDOVIC; EGYED; ROSENBLUM, 1999), (MEDVIDOVIC; TAYLOR, 2000), (MEI et al., 2002), (HILLIARD et al., 2012)

Tabela 4.3 – Tabela comparativa com os critérios estabelecidos entre ADLs

ADLs/ Criterios	**AADL**	**ABC/ADL**	**C2**	**COSA**	**Darwin**	**PRISMA**	**RAPIDE**	**SOFA**	**UniCon**	**Wright**
Referências [1]	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
Origem	CMU	PKU	UCI	Université de Nantes	Imperial College London	UPV	Stanford	Charles University	CMU	CMU
Publicação inicial	2004	2002	1995	2004	1993	2003	1993	1998	1996	1997
Geração	2	2	1	2	1	2	1	2	1	1
Tipo de domínio	Específico	Geral	Específico	Geral	Geral	Geral	Geral	Geral	Geral	Geral
Suporte a componentes	●	●	●	●	●	●	●	●	●	●
Suporte a conectores	●	●	●	●	○	●	○	●	●	●
Suporte a múltiplas visões arquiteturais	●	⊠	○	○	○	⊠	○	○	⊠	○
Suporte a padrões arquiteturais	⊠	●	◐	◐	◐	○	◐	●	●	●
Suporte a requisitos não funcionais	◐	◐	○	◐	○	●	○	◐	◐	○
Suporte a UML	●	●	○	●	○	●	○	●	○	○
Sintaxe textual	●	●	●	●	●	●	●	●	●	●
Sintaxe gráfica	●	●	●	●	●	●	●	●	●	○
Semântica formal	●	○	○	○	●	●	●	○	○	●
Geração de código	●	●	●	●	○	●	○	●	○	○
Amb. de desenvolvimento	●	●	●	●	●	●	●	●	●	⊠
Suporte a simulação	●	⊠	●	●	●	●	●	●	○	⊠
Referências [1] - Ver Tabela 4.2										

4.3 Decrição Resumida das ADLs

Nas Subseções a seguir são apresentadas, brevemente, descrições de cada ADL e como os critérios da Seção anterior são suportados para cada linguagem.

AADL

Em novembro de 2004 a *Society of Automotive Engineers* (SAE) lançou o AS5506, padrão aeroespacial chamado de *Architecture Analysis & Design Language* (AADL). AADL é uma linguagem de modelagem que suporta análise da arquitetura de um sistema com relação às propriedades críticas de desempenho por meio de uma notação extensível, *framework* e semântica definida com precisão.

AADL tem por objetivo analisar e especificar sistemas embarcados complexos de tempo real como sistemas aéreos, automotivos, industrial, médico, entre outros. AADL é considerada a ADL que possui a maior taxa de utilização por arquitetos (entre as ADLs, ou seja, a UML não consta na lista) devido a especialização em sistemas embarcados.

O código fonte dos componentes de software é fornecido pelo usuário e pode ser gerado a partir de geradores de componentes (como Matlab/Simulink ou Beacon), codificado manualmente ou por meio de reúso e reengenharia de software. As ferramentas podem automatizar o código fonte e o ambiente de execução, configuração do sistema, composição e geração do sistema em tempo de execução. Estas ferramentas de integração/geração de sistemas precisam ser consistentes com os métodos de análise e a semântica da AADL, para que os protótipos possam ser desenvolvidos rapidamente. Além disso, existe a AADS, uma ferramenta de simulação AADL que permite a verificação de requisitos funcionais e não funcionais dos protótipos do sistema.

As descrições AADL podem ser expressas textualmente, graficamente, em uma combinação dessas representações ou como XML. A AADL suporta visões arquiteturais para descrição do modelo.

A OMG desenvolveu em colaboração com a SAE o *profile* MARTE para fornecer os benefícios da AADL à comunidade UML. MARTE permite aos desenvolvedores usar ferramentas baseadas em UML para criar modelos de arquitetura de software com semântica formal. MARTE é um *profile* da UML para sistemas embarcados que usa semântica e conceitos da AADL. MARTE inclui elementos para representar o tempo por meio de relógios e temporizadores. Estes elementos podem ser usados para modelar o comportamento da arquitetura de software (GROUP et al., 2008), (FEILER et al., 2009).

ABC/ADL

ABC/ADL é uma ADL que suporta desenvolvimento de software baseado na composição de componentes. ABC/ADL suporta geração de código usando regras de mapeamento e

conectores customizáveis e estilos arquiteturais. A linguagem fornece construções abstratas para definir modelos e estilos arquiteturais. A arquitetura ABC/ADL é formada por um grupo de componentes e conectores interligados que cumprem as restrições impostas pelos estilos arquiteturais. Os padrões arquiteturais *blackboard, Aspect-oriented Programming* (AOP), *Subject-oriented Programming* (SOP) são exemplos de estilos encontrados na literatura.

ABC/ADL suporta parcialmente requisitos não funcionais, como segurança. ABC/ADL adota uma linguagem natural como sintaxe para facilitar o entendimento e uma sintaxe visual para facilitar o design. Em ABC/ADL, uma semântica formal não é estabelecida. A linguagem tem uma tecnologia de componentes de software que inclui três processos inter-relacionados: desenvolvimento, gerenciamento e composição de componentes.

O ambiente de desenvolvimento, ABC Tool, é um conjunto de ferramentas de suporte para ABC/ADL. A ferramenta ABC/ADL pode adicionar automaticamente métodos comuns na geração de novos componentes. ABC/ADL pode ser mapeada da descrição arquitetural para UML, gerar código Java e IDL a partir da descrição. Além disso, ele pode construir automaticamente as aplicações a partir de componentes existentes com base em especificações COTS de middleware, incluindo CORBA e J2EE.

C2

C2 é uma ADL focada nas descrições arquiteturais de aplicações orientadas a eventos. Os componentes de C2 operam simultaneamente, e se comunicam via troca de mensagens.

A interface de componentes e conectores consiste em mensagens de entrada e saída enviadas através de portas individuais. A semântica é expressa em termos de relações causais entre as mensagens de entrada e saída na interface do componente. Os conectores são modelados explicitamente e podem ser reutilizados. C2 possui sintaxe textual e gráfica, mas não define uma semântica formal e não oferece suporte a requisitos não funcionais.

C2 oferece suporte ao desenvolvimento C++, Ada e Java. C2 não suporta múltiplas visões arquiteturais, pois a ferramenta de desenvolvimento C2 chamada Argo apenas fornece a visão de desenvolvimento (gráfica e textual). C2 permite visualização do comportamento de uma arquitetura em execução e fornece ferramentas para visualização e filtragem de eventos gerados pela simulação.

COSA

COSA é uma ADL de propósito geral que combina os princípios da engenharia de software baseada em componentes com o paradigma de orientação a objetos (por exemplo, herança). O princípio básico da COSA é basear-se no formalismo das linguagens de descrições arquiteturais estendendo conceitos e mecanismos orientados a objetos para especificar arquiteturas de software. Em COSA, configurações, componentes e conectores são definidos como

classes que podem ser instanciadas para definir quaisquer arquiteturas diferentes. Os elementos básicos da arquitetura COSA são componentes, conectores, configurações, interfaces, restrições, requisitos funcionais e não funcionais.

O perfil COSA-UML é definido na UML 2.0 para o Eclipse 3.1. COSA-UML suporta a criação e gestão de modelos UML 2.0 para aplicações de software, independente da linguagem de programação. As descrições arquiteturais implementadas com COSA podem ser compatíveis com outras linguagens e ferramentas de descrição de arquitetura existentes com pouco ou nenhum esforço adicional do desenvolvedor.

COSA possui uma única visão arquitetural para configuração, componente e conector e não fornece uma semântica formal. O ambiente de desenvolvimento Cosastudio modela graficamente a arquitetura e auxilia os arquitetos na verificação da estrutura de um dado sistema.

Darwin

Darwin é uma das primeiras ADLs, originalmente desenvolvida como uma linguagem de configuração para a construção de um projeto chamado REX, cujo objetivo era permitir a implementação de componentes em uma variedade de linguagens de programação.

Darwin não suporta conectores. Por este motivo, os conectores não podem ser manipulados durante o projeto ou reutilizados no futuro. Como consequência, o suporte para estilos arquitetônicos ainda é limitado, devido a este conceito fraco de conectores.

Darwin permite que programas paralelos possam ser construídos a partir de descrições de componentes e suas interconexões. Esta linguagem não oferece suporte a UML, nem a requisitos não funcionais. Darwin possui uma semântica precisa, pois usa π-calculus para formalizar a semântica, define uma única visão arquitetural, fornecendo a descrição do modelo adequado apenas para a especificação de componentes e conectores.

Os componentes são implementados em uma linguagem de programação, a descrição arquitetural serve apenas para assegurar a interconexão adequada e comunicação entre eles, assim como a ADL UniCon. A ferramenta “Rex Designer” é usada para modelar a arquitetura graficamente.

PRISMA

PRISMA é uma ADL orientada a aspectos que visa a combinação de engenharia de software baseada em componentes com a engenharia de software orientada a aspectos. Esta linguagem possui interfaces, aspectos, componentes e conectores como elementos principais. As interfaces e os aspectos aumentam a capacidade de reutilização porque uma única interface pode ser utilizada por aspectos e somente um aspecto pode ser utilizado por diversos tipos de elementos. Além disso, componentes, conectores e sistemas podem ser reutilizados em diferentes modelos de arquitetura.

PRISMA é baseada em uma linguagem de especificação formal orientada a objetos chamada OASIS. OASIS é uma linguagem formal para definir modelos conceituais de sistemas orientados a objetos, que permite validar e gerar aplicações automaticamente a partir da informação capturada nos modelos. PRISMA utiliza OASIS para definir a semântica de modelos de arquitetura de uma maneira formal.

PRISMA gera automaticamente código fonte e especifica requisitos não funcionais. PRISMA possui sintaxe textual, sintaxe gráfica e pode ser especificada por XML. Ela suporta UML por meio do perfil PRISMA-UML e tem uma ferramenta PRISMANET para modelar e descrever uma arquitetura de software.

RAPIDE

RAPIDE é uma ADL de propósito geral com ênfase em simulação. Ela descreve e simula o comportamento da arquitetura de sistemas distribuídos, tem o objetivo de representar o estilo arquitetural baseado em eventos e modela comportamentos semelhantes a uma máquina de estados.

Esta ADL apresenta as capacidades de arquitetar, analisar, simular e gerar código, mas não suporta conectores, os conectores não podem ser identificados ou reutilizados, consequentemente, RAPIDE não permite conexões complexas de conectores-componentes, limitando a capacidade de descrever aplicações.

RAPIDE fornece apenas uma única visão arquitetural, adequada a especificação de componentes e conectores. O padrão arquitetural orientado a eventos é usado para definir a semântica precisa. RAPIDE fornece ferramentas para visualização e filtragem de eventos. Então, o arquiteto pode visualizar a simulação executável de uma arquitetura.

Em RAPIDE, os componentes e conectores possuem um alto nível de abstração e não prescrevem uma relação particular entre a descrição e a implementação da arquitetura. RAPIDE é usada apenas como linguagem de modelagem, não oferece suporte para implementação do modelo.

SOFA

SOFA tem foco em componentes, sua formalização e conectores. Ela pode dar suporte a estilos arquiteturais incluindo *procedure call, messaging, streaming e blackboard*. Atualmente, SOFA está na versão 2.0.

Na versão SOFA 2.0 a semântica é baseada em meta-modelo. Mais especificamente, a tecnologia MOF é usada para projetar um modelo de componente. Esta abordagem tem vantagens, como a geração automática de um repositório com interface padronizada, formato padronizado baseado em XML, suporte a geração automática de modelos, geração de código para componentes e configuração de uma aplicação.

Os conectores são responsáveis pela lógica de comunicação, o uso de conectores em SOFA possibilita diferentes estilos de comunicação e requisitos não funcionais como *logging, benchmarking*, verificação de comportamento em tempo de execução, segurança, entre outros. Além disso, ela se concentra na geração automática de código a partir das especificações do conector.

SOFA oferece suporte apenas a uma visão arquitetural, a visão de implementação. SOFA2-UML é um conjunto de plugins do Eclipse que se estende na IDE de SOFA2, estes plugins tem capacidade de gerar modelos de componentes UML, com isso o mapeamento das entidades entre UML e SOFA 2.0 podem ser facilmente especificados.

UniCon

UniCon é uma ADL de propósito geral com ênfase em geração de conectores. UniCon tem foco em apoiar os estilos de arquitetura e a geração de código a partir da especificação das descrições da arquitetura. Esta ADL permite aos arquitetos especificar componentes e conectores. Além disso, oferece um conjunto de ferramentas para mapear a arquitetura em código fonte na linguagem C e permite também a simulação do sistema.

UniCon fornece suporte para a modelagem de requisitos não funcionais, tem sintaxe gráfica e fornece ambiente de modelagem de arquitetura. No entanto, UniCon não oferece suporte a UML e não define uma semântica formal.

UniCon permite gerar sistemas executáveis a partir de descrições de arquitetura, desde que implementações de componentes já existam. Portanto, os componentes são implementados em uma linguagem de programação, e a descrição arquitetural serve apenas para garantir a conexão e comunicação adequada entre eles.

Wright

Wright é uma ADL de propósito geral com ênfase em análise de protocolos de comunicação. Wright pode analisar tanto a arquitetura dos sistemas de softwares individuais como de famílias dos sistemas. O projeto Wright se concentra em conectores explícitos, verificação automática de propriedades arquiteturais e formalização de estilos arquiteturais.

Em Wright, os componentes e conectores possuem um alto nível de abstração e não assumem ou prescrevem uma relação particular entre uma descrição de arquitetura e uma implementação. Portanto, Wright é estritamente usado como uma linguagem de modelagem e não fornece suporte para implementação do modelo.

Wright tem semântica formal e permite a especificação de diferentes estilos, mas não oferece suporte a UML, não tem sintaxe gráfica, não há suporte para requisitos não funcionais e define uma única visão arquitetural.

4.4 Discussão dos Resultados

Nesta Seção é apresentada a análise comparativa entre os critérios definidos para comparação entre as ADLs.

Foram reunidas dez (10) ADLs para realização da análise, das quais cinco (5) ADLs são de 1ª geração e cinco (5) ADLs são de 2ª geração. Todas as ADLs foram criadas por iniciativa de universidades no decorrer dos anos de 1993 a 2004. As únicas ADLs criadas para domínio específico foram AADL e C2, as demais oferecem suporte para domínio geral.

Mediante a apuração dos dados da pesquisa realizada e observada na Tabela 4.3, é correto afirmar que todas as ADLs oferecem suporte a componentes, possuem ambiente de desenvolvimento (exceto Wright que não foi especificado), possuem sintaxe textual e gráfica (exceto Wright que não tem sintaxe gráfica). Apenas Darwin e RAPIDE não oferecem suporte a conectores. A maioria das ADLs gera automaticamente o código fonte, mas não suportam múltiplas visões arquiteturais, a metade possui semântica formal e suporte a UML. Requisitos não funcionais e padrões arquiteturais são parcialmente descritos pela maioria das ADLs.

A maioria dos critérios foi cumprida pela AADL e PRISMA. Darwin e Wright tiveram as taxas mais baixas de critérios satisfeitos por esta análise. No entanto, nenhuma das linguagens analisadas contemplou todos os critérios estabelecidos.

De acordo com os autores em (MALAVOLTA et al., 2013), existem diversas linguagens desenvolvidas ao longo dos últimos anos, o que torna difícil para os arquitetos avaliarem e, em seguida, escolherem a ADL mais adequada para um projeto específico. Por exemplo, se existe uma grande equipe envolvida em um único projeto, e cada pessoa ou equipe tiver uma visão arquitetural diferente, então AADL é a melhor opção para descrição do projeto arquitetural. Considerando suas características intrínsecas, SOFA tem preocupação maior no suporte aos estilos arquitetônicos. Com a análise das ADLs pesquisadas, tornou-se possível classificá-las pelas principais preocupações e interesses. A lista completa de cada ADL é descrita na Tabela 4.4.

4.5 Contribuições do Capítulo

Este capítulo reúne e descreve uma análise comparativa entre 10 ADLs de acordo com 16 características encontradas para realizar a descrição arquitetural, com objetivo de mostrar a situação atual das linguagens e sua evolução ao longo dos anos. Além disso, são discutidos os progressos, problemas e considerações sobre a aplicação de ADLs na prática. As ADLs evoluíram e começaram a suportar UML, requisitos não funcionais e gerar código fonte automaticamente.

Um resultado importante encontrado nesta análise é que a maioria das ADLs não suporta múltiplas visões arquiteturais, o que é conhecido como uma boa prática na arquitetura de software. Outro fator importante descrito neste capítulo, foi indicar as principais preocupações e interesses

Tabela 4.4 – Principais preocupações e interesses das ADLs.

ADLs	**Preocupação**
AADL	Sistemas embarcados de tempo real
ABC/ADL	Composição de componentes
C2	Aplicações dirigidas a eventos
COSA	Combina engenharia baseada em componentes com programação orientada a objetos
Darwin	Implementação de componentes em linguagens de programação
PRISMA	Combina engenharia baseada em componentes com programação orientada a aspectos
RAPIDE	Simulação de sistemas distribuídos
SOFA	Estilos arquiteturais
UniCon	Geração de conectores
Wright	Análise de protocolos de comunicação

das ADLs pesquisadas. Esta análise é capaz de ajudar os arquitetos de software a escolherem uma ADL mais apropriada entre as demais. Além disso, há uma carência em pesquisas relacionadas ao reúso de software, incluindo estilos arquiteturais que suportam linhas de produtos de software e ADLs para suportar múltiplas visões arquitetônicas.

Nenhuma das linguagens analisadas contempla todos os critérios de análise comparativa. Considerando este resultado, será conduzido um estudo de caso em um campo multidisciplinar, como o projeto arquitetural de sistemas embarcados automotivos, que têm de lidar com restrições em tempo real. A ideia principal neste estudo de caso é usar UML para design de software e a SysML para descrição de arquitetura de software, combinando os modelos para apresentar uma técnica para modelar a arquitetura de sistemas embarcados de tempo real.

5 Técnica para Modelar a Arquitetura de Sistemas Embarcados de Tempo Real com SysML e UML

Este capítulo é baseado no artigo *"A Technique to Architect Real-Time Embedded Systems with SysML and UML through Multiple Views"* aceito na conferência *International Conference on Enterprise Information Systems* (ICEIS).

À medida que a complexidade dos sistemas eletrônicos e das aplicações embarcadas aumenta, há uma necessidade contínua de representações mais abstratas de tais sistemas (KHAN; MALLET; RASHID, 2015). Modelar a arquitetura de sistemas é uma tarefa desafiadora, pois esses sistemas não são apenas de grande magnitude, mas também são significativamente diversos (KHAN; MALLET; RASHID, 2015). Além disso, desafios e dificuldades ocorrem ao implementar esses sistemas, devido a características específicas, incluindo mobilidade, segurança ou restrições em tempo real (KOOPMAN, 2010), (MARQUES; SIEGERT; BRISOLARA, 2014).

Geralmente, os modelos são reconhecidos na engenharia como formas essenciais para compreensão dos sistemas complexos do mundo real (BROWN, 2004). Os modelos são usados para prever a qualidade do sistema, colaborar para o raciocínio das propriedades específicas quando os requisitos do sistema são alterados e comunicar as principais características do sistema aos seus *stakeholders*. Os modelos podem ser desenvolvidos como um precursor para implementar o sistema físico, podem ser derivados de um sistema existente ou ainda podem servir no auxílio do entendimento comportamental de um sistema em desenvolvimento (BROWN, 2004).

Um dos principais desafios a ser enfrentado no desenvolvimento de soluções é conectar os requisitos específicos de domínio expressos pelos analistas de negócios com as soluções específicas de tecnologia projetadas por arquitetos de software (BROWN, 2004), (AUWERAER et al., 2013). Normalmente, a conexão entre esses dois mundos diferentes é relativamente baixa, as duas comunidades têm habilidades diferentes, usam conceitos de modelagem e notações diferentes (se é que existem) e raramente entendem o mapeamento entre esses conceitos (BROWN, 2004), (AUWERAER et al., 2013), (CHIAO; KUNZLE; REICHERT, 2013).

Descrever a arquitetura de sistemas em tempo real por meio de linguagens semi-formais tem sido frequentemente considerado na literatura (PANDEY, 2010), (MALAVOLTA et al., 2013), (MELO; SOARES, 2014). No entanto, a abordagem mais comum é utilizar múltiplas linguagens de modelagem em fases separadas e de forma totalmente independente (MENS; STRAETEN; D'HONDT, 2006), (BLANC et al., 2008), (LUCAS; MOLINA; TOVAL, 2009),

(RUCHKIN; SCHMERL; GARLAN, 2016). Por exemplo, um robô pessoal precisa evitar colisões, selecionar planos eficientes de movimento a longo prazo e estar ciente de seus recursos. Para satisfazer os requisitos desse sistema robótico, os *stakeholders* podem utilizar modelos que variam em formalismo, nível de abstrações, e em representações das partes do sistema (RUCHKIN; SCHMERL; GARLAN, 2016). Esta situação nem sempre é possível e o pressuposto neste capítulo é propor uma técnica na qual diagramas de apenas duas linguagens de modelagem, UML e SysML, sejam usados em conjunto durante toda a fase de modelagem e documentação, além de servir na comunicação com os *stakeholders* em todas as fases do projeto de software.

A linguagem de modelagem de sistemas (SysML) oferece recursos adicionais à UML, incluindo modelagem de requisitos (MARQUES; SIEGERT; BRISOLARA, 2014). SysML também tem recursos para suportar a especificação de diversos aspectos estruturais, comportamentais e temporais de sistemas embarcados complexos (KHAN; MALLET; RASHID, 2015). Portanto, a técnica proposta é capaz de modelar tanto o software quanto os elementos arquiteturais do sistema, atendendo aos critérios de suporte a modelagem de componentes e conectores do modelo, sintaxe gráfica e textual, modelagem de requisitos não funcionais, *design* da visão estrutural de software usando classes da UML, além de representar elementos de hardware na arquitetura e descrever a rastreabilidade entre requisitos. Um estudo de caso sobre um sistema embarcado automotivo em tempo real é apresentado para ilustrar a técnica proposta.

5.1 Modelagem do Sistema de Controle de *Airbag*

No Capítulo 3, Seção 3.2, foram descritos os requisitos textuais do sistema de controle de *airbag* de acordo com a Norma ISO/IEC/IEEE 29148:2011. Neste capítulo, são apresentados os diagramas da SysML e o diagrama de Classes da UML que são usados para a modelagem do sistema de controle de *airbag*, e a técnica proposta com a união das duas linguagens.

5.1.1 Diagrama e Tabela da SysML para o Sistema de Controle de *Airbag*

O diagrama de Requisitos da SysML do *airbag* é ilustrado na Fig. 5.1. O diagrama inclui uma descrição de texto reduzida, por motivos de espaço (Ver texto completo no Capítulo 3, Seção 3.2).

O diagrama de Requisitos reúne os requisitos que tenham relação e decompõe um requisito complexo em requisitos mais simples e únicos. O relacionamento *"DeriveRqt"* relaciona um requisito derivado com o seu requisito de origem. Isso normalmente envolve análise para determinar os requisitos derivados de um requisito de origem. Os requisitos derivados geralmente correspondem a requisitos no próximo nível da hierarquia do sistema. O relacionamento *"Containment"* (representado pelo símbolo $\oplus$—— no diagrama da Fig. 5.1) especifica a hierarquia entre requisitos, seu uso impede a reutilização de requisitos em diferentes contextos, uma vez que um determinado elemento de modelo só pode existir em um *"Containment"*. O relacionamento

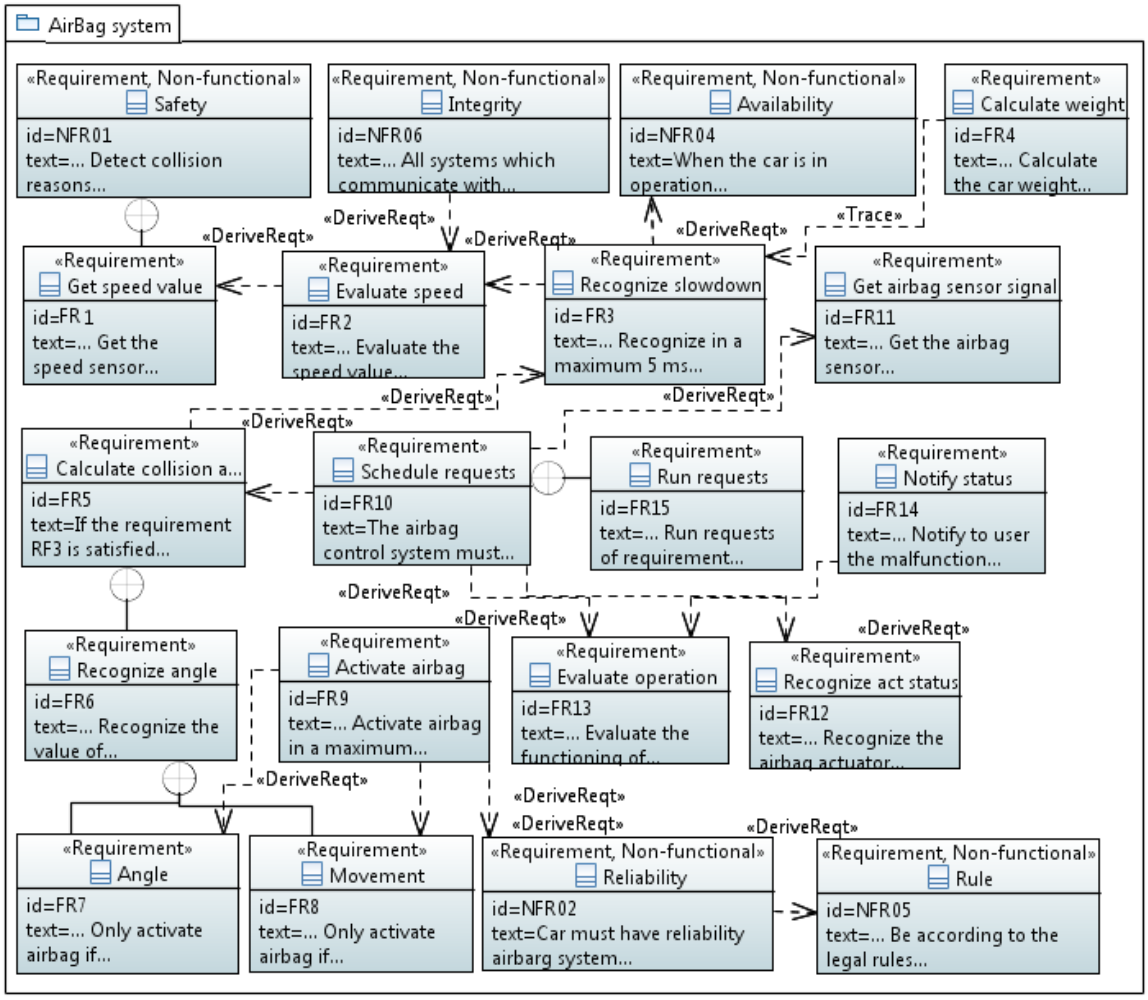

Figura 5.1 – Diagrama de Requisitos (SysML) do sistema de *airbag*.

"Trace" é um relacionamento de propósito geral entre um requisito e qualquer outro elemento do modelo, seu uso é apenas por razões de rastreabilidade. Ele expressa que os elementos clientes do modelo têm um relacionamento com um requisito fornecedor, mas sem especificar essa relação em mais detalhes (OMG, 2015).

Com o objetivo de modelar o conceito de requisitos não funcionais, foi especificado um estereótipo chamado ⟨⟨ Non-functional ⟩⟩. Este estereótipo permite a modelagem de requisitos não funcionais e a distinção com os demais, facilitando a compreensão dos requisitos e composição destes na arquitetura. Os estereótipos da SysML definem novas construções de modelagem estendendo construções existentes da UML 2 com novas propriedades e restrições (OMG, 2015). Os usuários da SysML podem personalizar as taxonomias de requisitos definindo subclasses adicionais do estereótipo de requisito, por exemplo, os usuários podem definir categorias de requisitos para representar requisitos não funcionais, de interface, de desempenho, de ambiente físico, ambiente de armazenamento, restrições de projeto, entre outros requisitos especializados.

A SysML também permite a representação de requisitos, suas propriedades e relacionamentos em um formato tabular. Os campos da tabela são o ID, nome e tipo do requisito. Nesta

proposta, foram adicionados os campos *"derived Req"* representando os requisitos derivados. Citando como um exemplo, o requisito que possui o ID FR5 é um requisito fornecedor do requisito FR10, por sua vez, o FR10 é um requisito derivado/cliente de FR5. O campo *"derived-From Req"* representando que o primeiro requisito foi derivado a partir de outro, voltando ao exemplo, o requisito com ID FR5 foi derivado a partir do requisito FR3, ou seja, FR5 é cliente do FR3. O campo *"containment Req"* representa a hierarquia entre requisitos, no exemplo em questão, o FR6 é um subrequisito do FR5, ou seja, FR6 é filho do FR5. Todos estes campos e relacionamentos podem ser visualizados na Tabela 5.1.

Tabela 5.1 – Tabela de Requisitos (SysML) do sistema de *airbag*.

id	name	type	derived Req	derivedFrom Req	containment Req
FR8	Movement	functional	FR9	-	-
FR7	Angle	functional	FR9	-	-
FR9	Activate *airbag*	functional	-	NFR02, FR8, FR7	-
FR1	Get speed value	functional	FR2	-	-
FR4	Calculate weight	functional	FR3	-	-
FR2	Evaluate speed	functional	FR3, NFR06	FR1	-
FR3	Recognize slowdown	functional	FR5, FR4	FR2, NFR04	-
FR5	Calculate collision angle	functional	FR10	FR3	FR6
FR6	Recognize angle	functional	-	-	FR7,FR8
FR11	Get *airbag* sensor signal	functional	FR10	-	-
FR12	Recognize act status	functional	FR10	-	-
FR10	Schedule requests	functional	-	FR12, FR11, FR13, FR5	FR15
FR14	Notify status	functional	-	FR13	-
FR13	Evaluate operation	functional	FR10, FR14	-	-
NFR01	Safety	non-functional	-	-	FR1
NFR02	Reliability	non-functional	FR9	NFR05	-
NFR04	Availability	non-functional	FR3	-	-
NFR05	Rule	non-functional	NFR02	-	-
NFR06	Integrity	non-functional	-	FR2	-
FR15	Run requests	functional	-	-	-

5.1.2 Diagrama de Definição de Bloco para o Sistema de Controle de *Airbag*

No exemplo, o BDD foi usado para modelar componentes envolvidos no sistema de *airbag*. As funções especificadas são implementadas para comunicações seguras com outros componentes e com sensores/atuadores locais. O BDD da SysML é representado na Fig. 5.2. Funções de *airbag* frequentemente envolvem comunicações entre si e, consequentemente, têm uma arquitetura distribuída complexa. Estas funções são implementadas por uma unidade de controle eletrônico (ECU) chamada controle de comunicação, que suporta a recepção de solicitações enquanto as outras unidades executam suas próprias ações locais.

Um BDD da SysML pode descrever elementos estruturais gerais, variando de pequeno a muito grande. Os Blocos da SysML podem ser usados também para representar a arquitetura física do sistema (OMG, 2015). No exemplo do *airbag*, 12 blocos relacionados são definidos. O relacionamento entre blocos foi representado por associações de composição. A instância de composição é síncrona, ou seja, se uma instância for destruída, finalizando a execução, as outras execuções também serão encerradas. O bloco *"Clock"* controla a sincronia de tempo. O bloco

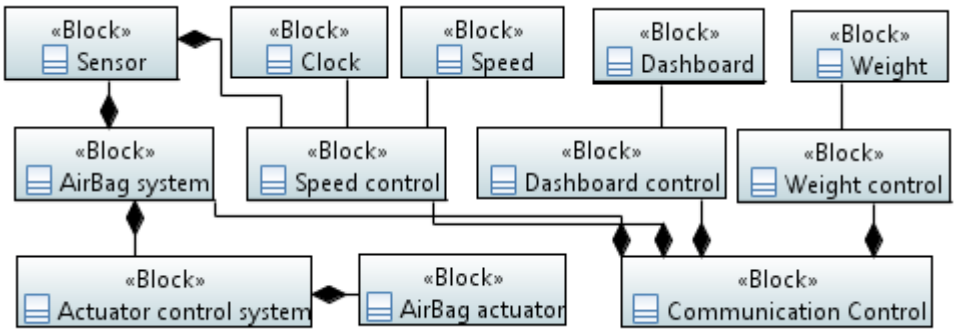

Figura 5.2 – Diagrama BDD (SysML) do sistema de *airbag*.

"Airbag system" comunica com outros blocos que não fazem parte diretamente do sistema de *airbag* por meio do bloco *"Communication control"*. Essa comunicação é essencial para que as informações necessárias à execução do sistema de *airbag* sejam conduzidas. As informações são: obter a velocidade do veículo a cada 15 milissegundos, informação essa obtida pelo bloco *"Speed control"*, obter o peso do veiculo e dos passageiros pelo bloco *"Weight control"*, transmitir informações de mal funcionamento do *airbag* para o bloco *"Dashboard control"*. Além disso, o sistema de *airbag* transmite e recebe dados do sensor e atuadores, que diretamente fazem parte do sistema de *airbag*.

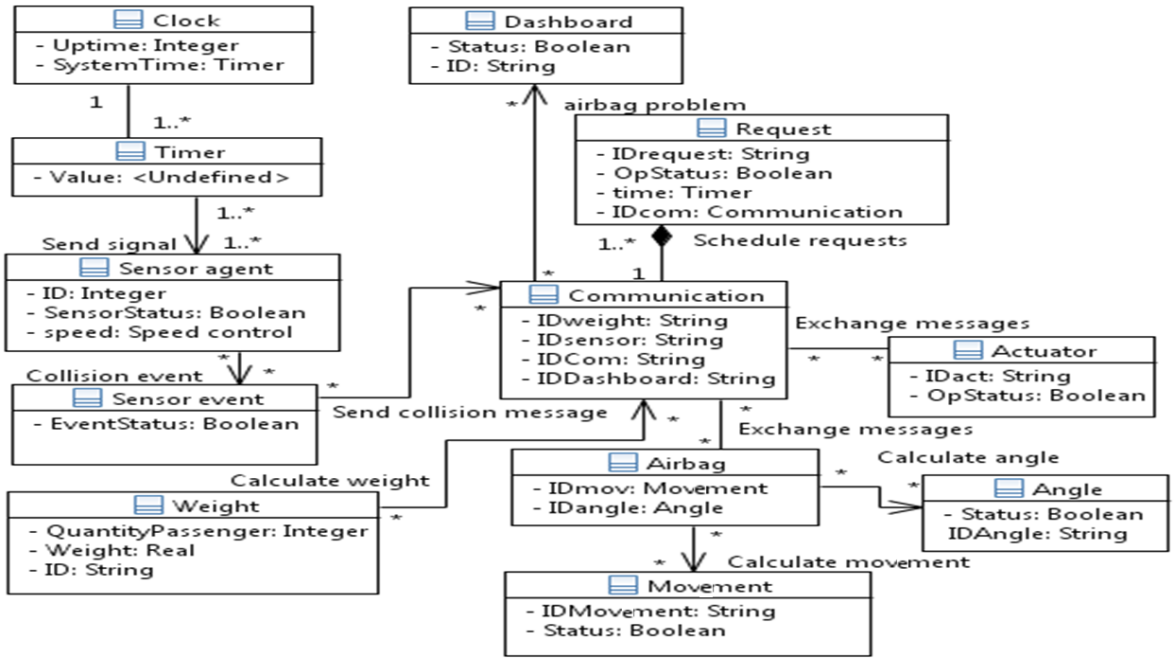

Figura 5.3 – Diagrama de Classes (UML) do sistema de *airbag*.

Com o BDD, o hardware, os dados e os procedimentos podem ser modelados. A representação da arquitetura pode ser feita por meio de blocos com foco na estrutura geral do software, incluindo partes de cada bloco, restrições e propriedades não necessariamente relacionadas ao software. Além disso, os Blocos da SysML são candidatos a serem refinados em uma ou mais

Classes da UML (ver Fig. 5.3) durante as fases de projeto e implementação do software.

5.1.3 Diagrama de Bloco Interno para o Sistema de Controle de *Airbag*

O IBD da SysML (Fig. 5.4) modela as operações internas do bloco *"AirBag System"*. As portas se conectam a entidades externas e interagem com um bloco por meio de conectores. Uma porta de fluxo especifica os itens de entrada e saída que podem fluir entre um bloco e seu ambiente. As portas de *airbag* especificam o caminho do fluxo, comunicação de dados, operações, recepções e recursos de hardware (BUS). Para modelar um caminho de fluxo, foi utilizado *"FlowPort"* especificando os itens de entrada e saída que podem fluir entre um bloco e seu ambiente. A notação da porta de fluxo é um quadrado no limite do bloco proprietário e tem uma seta dentro do limite. As portas de fluxo retransmitem os itens de entrada e saída para um conector que conecta blocos às partes internas de seu proprietário. Os conectores são rotulados por mensagens que representam operações entre duas partes.

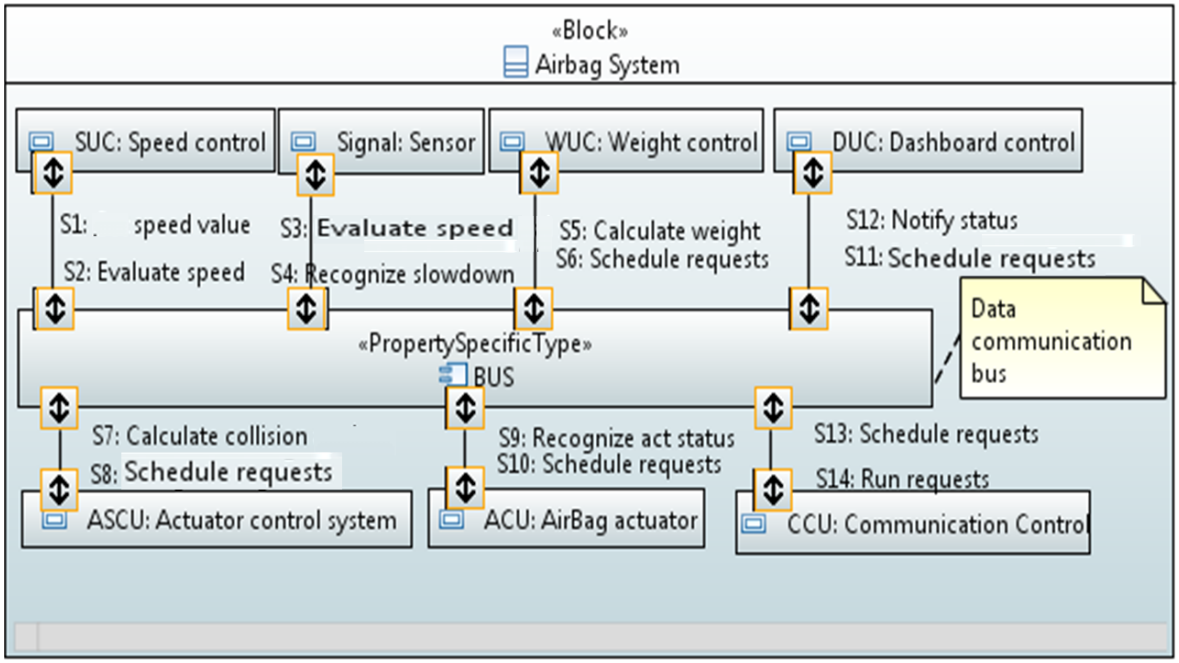

Figura 5.4 – Diagrama IBD (SysML) do sistema de *airbag*.

5.1.4 Diagrama de Atividades para o Sistema de Controle de *Airbag*

A atividade principal do *airbag* está sendo representada pelo diagrama de Atividades na Fig. 5.5. O diagrama apresenta a atividade de ativação de *airbag*. Foram modeladas as atividades inicial e final, o nó de decisões que pode ter duas possíveis saídas, o loop iterativo com ⟨⟨*loop node*⟩⟩, representando a repetição das funções do sistema, restrições de tempo e ângulo com a linguagem *Object Constraint Language* (OCL). OCL é uma linguagem padrão da UML para especificação de restrições em objetos. Cada restrição formal consiste em uma descrição textual expressa em OCL (OMG, 2015).

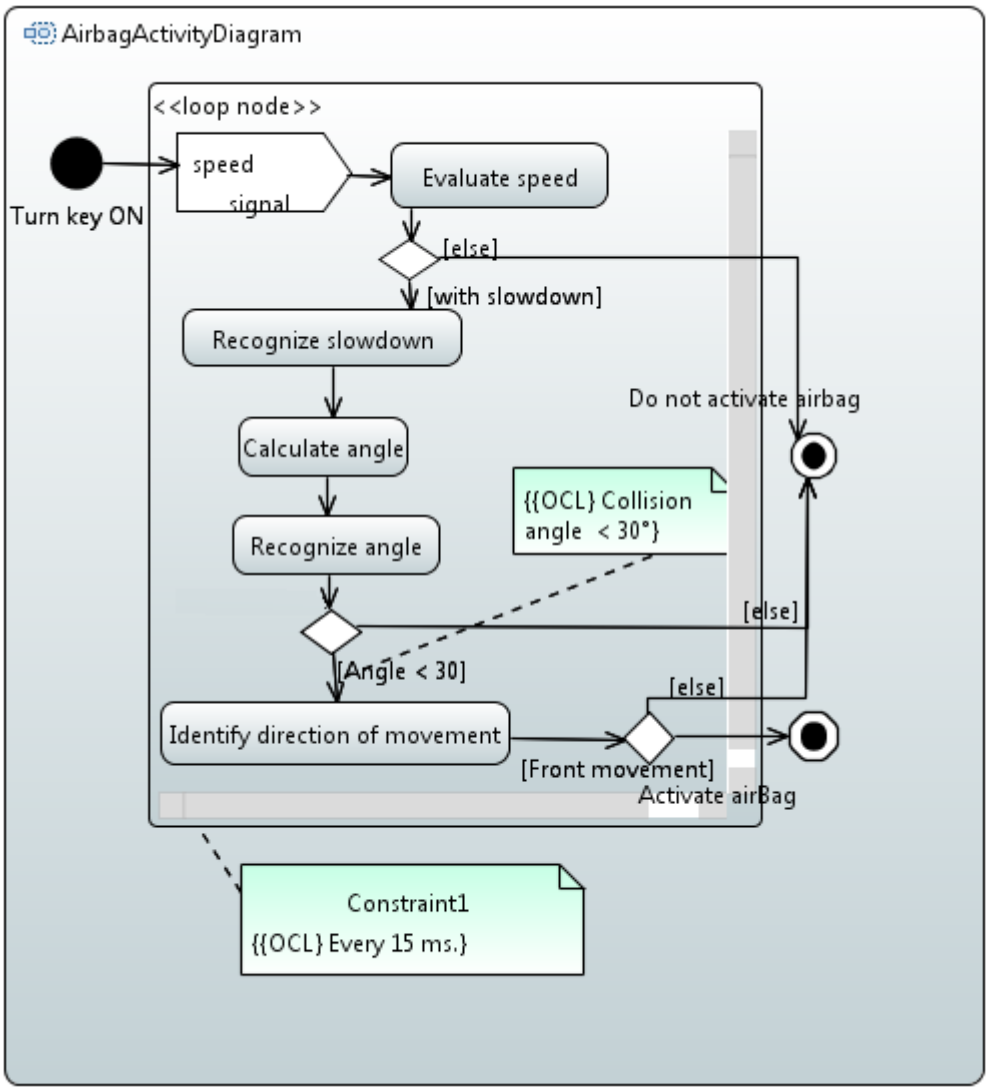

Figura 5.5 – Diagrama de Atividades (SysML) do sistema de *airbag*.

5.2 Técnica proposta combinando SysML e UML

A técnica tem foco em modelos projetados ao longo de três visões arquiteturais: requisitos, estrutura e comportamento. O diagrama de Requisitos da SysML ilustrado na Fig. 5.1 e a tabela de Requisitos da SysML mostrada na Tabela 5.1 foram utilizados para expressar os requisitos. A estrutura do sistema é evidenciada no diagrama BDD e no diagrama IBD, ambos diagramas da SysML, ilustrado nas Figs. 5.2 e 5.4, respectivamente. Finalmente, o diagrama de Atividades da SysML, ilustrado na Fig. 5.5 é empregado para capturar o comportamento do sistema. Esta técnica é aplicada para relacionar os modelos e analisar o impacto dos requisitos tanto no software quanto no projeto do sistema embarcado.

Na Tabela 5.1, é possível fazer uma análise de como uma mudança em um requisito impactará nos outros requisitos relacionados, por meio dos campos *derived Req, derived from Req e containment Req*.

Para perceber esse impacto, é necessário compreender o conceito dos relacionamentos entre requisitos. O relacionamento *"Derive"* relaciona um requisito derivado ao seu requisito de origem. Estes requisitos derivados geralmente correspondem aos requisitos no próximo nível da hierarquia do sistema (OMG, 2015). No relacionamento *"containment"*, um requisito composto pode conter sub-requisitos, que descreve uma árvore de hierarquia de requisitos. Este relacionamento permite que um requisito complexo seja decomposto em seus requisitos filhos (OMG, 2015).

Tabela 5.2 – Requisitos associados a um bloco particular.

ID	SATISFY (BLOCK)
FR8	*airbag* system
FR7	*airbag* system
FR9	actuator control system
FR1	speed control
FR4	weight control
FR2	speed control
FR3	sensor
FR5	*airbag* system
FR6	*airbag* system
FR11	sensor
FR12	actuator control system
FR10	communication control
FR14	dashboard control
FR13	*airbag* actuator
FR15	communication control

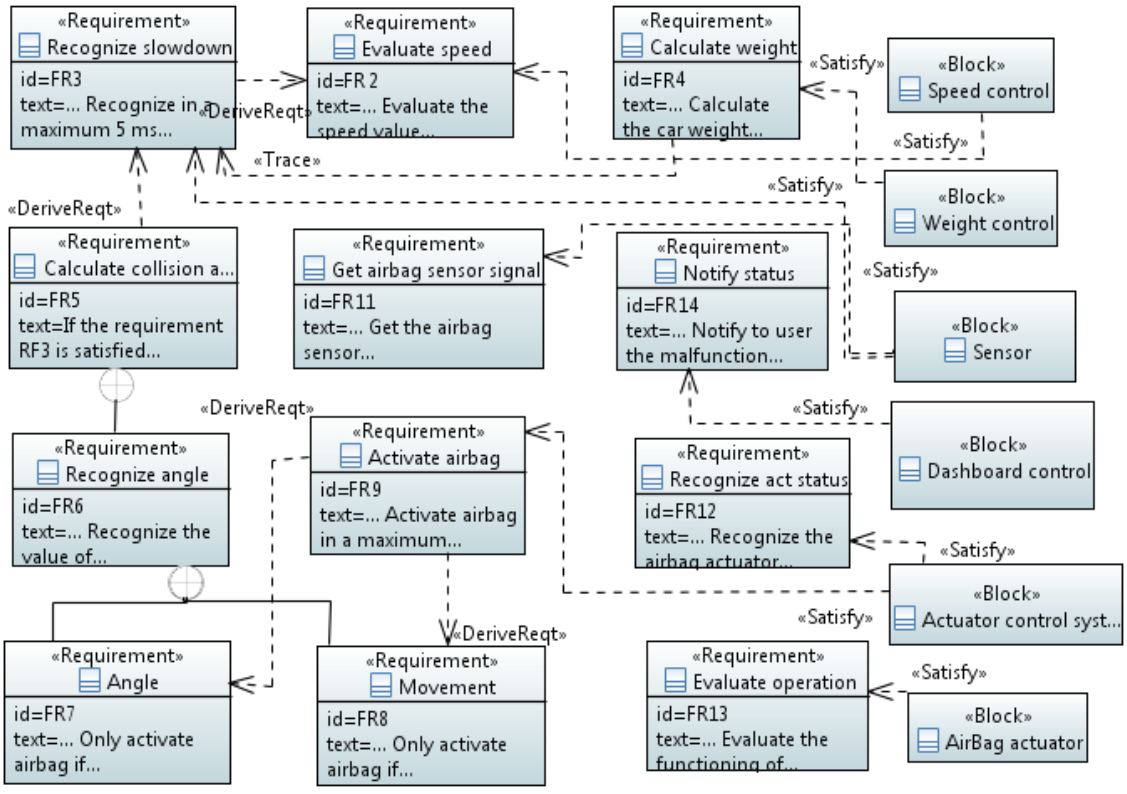

Figura 5.6 – Diagrama de Requisitos combinado com o diagrama BDD (SysML).

Tabela 5.3 – Requisitos associados a um bloco particular.

id	name	type	derived Req	derivedFrom Req	containment Req	Satisfy (BLOCK)
FR8	Movement	functional	FR9	-	-	*airbag* system
FR7	Angle	functional	FR9	-	-	*airbag* system
FR9	Activate *airbag*	functional	-	NFR02, FR8, FR7	-	actuator control system
FR1	Get speed value	functional	FR2	-	-	speed control
FR4	Calculate weight	functional	FR3	-	-	weight control
FR2	Evaluate speed	functional	FR3, NFR06	FR1	-	speed control
FR3	Recognize slowdown	functional	FR5, FR4	FR2, NFR04	-	sensor
FR5	Calculate collision angle	functional	FR10	FR3	FR6	*airbag* system
FR6	Recognize angle	functional	-	-	FR7,FR8	*airbag* system
FR11	Get *airbag* sensor signal	functional	FR10	-	-	sensor
FR12	Recognize act status	functional	FR10	-	-	actuator control system
FR10	Schedule requests	functional	-	FR12, FR11, FR13, FR5	FR15	communication control
FR14	Notify status	functional	-	FR13	-	dashboard control
FR13	Evaluate operation	functional	FR10, FR14	-	-	*airbag* actuator
NFR01	Safety	non-functional	-	-	FR1	All blocks
NFR02	Reliability	non-functional	FR9	NFR05	-	All blocks
NFR04	Availability	non-functional	FR3	-	-	All blocks
NFR05	Rule	non-functional	NFR02	-	-	All blocks
NFR06	Integrity	non-functional	-	FR2	-	All blocks
FR15	Run requests	functional	-	-	-	communication control

Neste trabalho, a primeira fase para a construção da técnica proposta é relacionar os requisitos a seus respectivos blocos. O relacionamento entre os blocos e os requisitos da SysML é mostrado no diagrama da Fig. 5.6. O relacionamento *"satisfy"* deste diagrama descreve como um bloco satisfaz um ou mais requisitos, esta relação representa dependência de um requisito a um sistema particular. A associação de um requisito ao seu bloco também foi desenvolvida em formato tabular, a técnica proposta é apresentada na Tabela 5.2 . Assim, é possível identificar quais requisitos estão associados a um bloco/sistema particular e também identificar qual requisito pertence a qual bloco.

O campo *"satisfy (BLOCK)"* relacionando um bloco a um requisito está incluído na Tabela 5.3. A Tabela 5.3 contém informações completas dos requisitos e dos blocos. A percepção de modelos relacionados é expandida quando comparada com a forma como os requisitos e blocos são expressos usando a linguagem tabular. Além dos Requisitos e Blocos da SysML serem descritos, as informações de rastreabilidade representadas pela técnica são fornecidas, por exemplo, quando um requisito é modificado, é possível rastrear quais requisitos e blocos são afetados. Os requisitos não funcionais estão incluídos em todos os blocos porque todos os blocos devem estar em conformidade com os requisitos não funcionais. Usando esta técnica, os profissionais de sistemas e de software podem se comunicar sobre o projeto com mais facilidade.

A segunda fase da técnica é baseada na combinação do diagrama de Classes da UML ao BDD da SysML por meio da conexão *"dependency"* na Fig. 5.7. *"Dependency"* é um

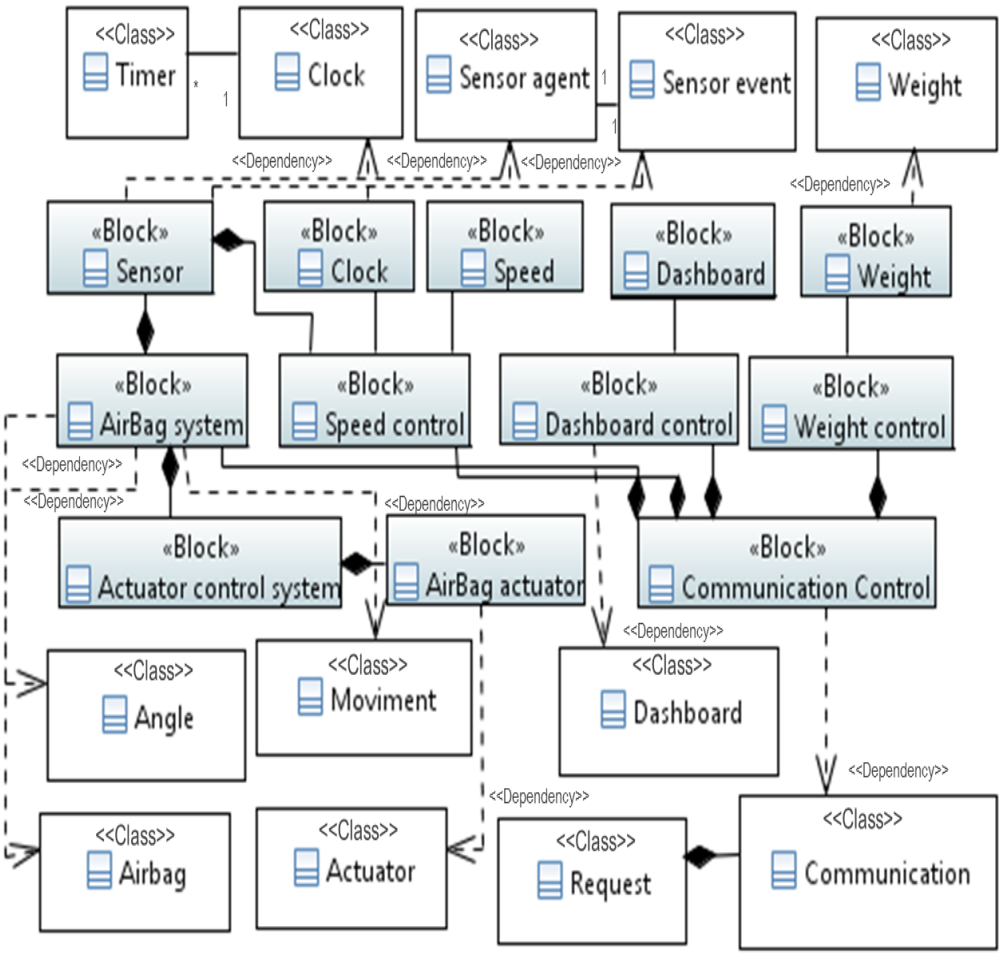

Figura 5.7 – Diagrama de Classes (UML) combinado com diagrama BDD (SysML).

relacionamento simples entre bloco e classe. A principal razão de modelar conjuntamente os componentes com o diagrama de Blocos da SysML e diagrama de Classes da UML se deve ao fato que esta técnica fornece uma maneira simples de mapear elementos de sistemas para elementos de software. Além disso, conhecer o mapeamento de elementos de sistema para elementos de software pode reunir o trabalho de *stakeholders* do sistema e do software. Normalmente, as dificuldades de entender domínios de negócios altamente complexos são tipicamente combinadas com desafios de gerenciar o esforço de desenvolvimento envolvendo grandes equipes em variadas fases de um projeto que abrange meses. As pressões de tempo no mercado inerentes aos esforços de desenvolvimento dos produtos servem para agravar os problemas (BROWN, 2004).

A terceira fase consiste em combinar o diagrama de Atividades com o diagrama de Requisitos, ambos da SysML, conforme ilustrado na Fig. 5.8. Nesta fase é possível analisar como uma alteração em um requisito afetará a atividade desenvolvida, quais ações serão satisfeitas por um requisito e também quais ações serão afetadas se ocorrer uma mudança neste mesmo requisito. Neste caso, o comportamento do sistema é expresso em consistência com os requisitos associados.

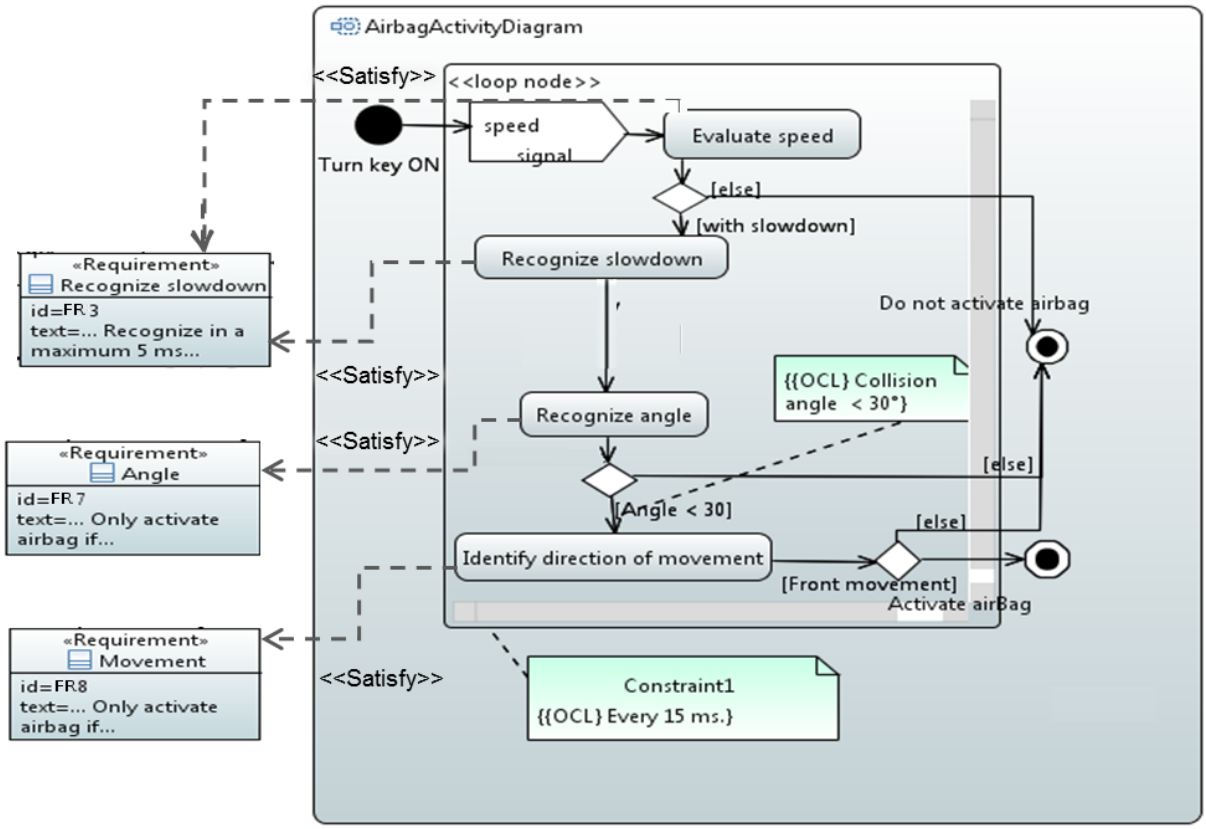

Figura 5.8 – Diagrama de Requisitos e diagrama de Atividades (SysML).

5.3 Comparação com trabalhos relacionados

Recentemente, a combinação de linguagens com SysML tem sido investigada para sistemas embarcados e complexos, a fim de suportar projetos de sistemas. Em Riccobene e Scandurra (2012), os autores apresentam a integração de dois perfis da UML: SysML e SystemC. Em outra pesquisa, os autores Melo e Soares (2014) combinam o diagrama de Blocos da SysML com o diagrama de Classes da UML para projetar a visão estrutural de uma arquitetura para *software-intensive systems*. No trabalho de Vogel-Heuser et al. (2014) foi apresentado um experimento com UML e SysML sobre aspectos de usabilidade. A intenção era mostrar o aumento da eficiência e qualidade do software no processo de desenvolvimento. Uma abordagem de engenharia de requisitos baseada em modelos para o domínio de software embarcado é descrita em Marques, Siegert e Brisolara (2014). A abordagem é baseada nas notações UML, MARTE e SysML, que são integradas para melhorar a especificação de requisitos e rastreabilidade. Os autores se preocuparam com a necessidade de um procedimento para modelar/especificar requisitos funcionais e não funcionais e garantir que esses requisitos tenham sido satisfeitos.

Outros artigos investigam como relacionar SysML com linguagens diferentes do contexto da UML e do padrão *Meta Object Facility* (MOF), como em Giese, Hildebrandt e Neumann (2010), os autores mostram como as regras de sincronização e consistência de modelos podem ser aplicadas para automatizar modelos da SysML com modelos descritos com a linguagem

AUTOSAR. O objetivo é garantir que os diferentes modelos possam ser mantidos consistentes. Autores do trabalho Behjati et al. (2011) propuseram o perfil ExSAM que estende SysML adicionando conceitos da AADL para ter a vantagem de usar ambas as linguagens em forma contínua. No entanto, AADL e SysML têm semânticas diferentes, e ocorre sobreposições conceituais entre as duas linguagens, tais como modos/máquinas de estado, componentes/blocos de sistema e componentes de interações/fluxo de blocos. Além disso, escolher entre usar AADL ou SysML para modelagem de relacionamentos é, por vezes difícil, por exemplo, na SysML, "*realization*" permite a reutilização de atributos de componentes físicos, e em AADL, os atributos físicos não podem ser manipulados diretamente.

Em Khan, Mallet e Rashid (2015) é proposta uma abordagem unificada para o projeto de um sistema e sua verificação baseada em SysML, MARTE, *Clock Constraint Specification Language* (CCSL) e SystemVerilog com o intuito de ajudar engenheiros de hardware no processo de verificação. No entanto, combinar três linguagens de modelagem (SysML, MARTE, CCSL) e mais uma linguagem de hardware (SystemVerilog) pode ser confuso para engenheiros, precisa de esforços adicionais, tempo e custo para gerenciar o processo.

Em contraste com outros trabalhos, a técnica definida e apresentada neste trabalho tem como foco combinar UML e SysML a fim de comunicar o mesmo projeto com diferentes *stakeholders* envolvidos. Nesta técnica são mostrados: a rastreabilidade entre requisitos, o projeto estrutural combinando o diagrama de Requisitos da SysML com o diagrama de Blocos da SysML e o diagrama de Blocos da SysML interagindo com o diagrama de Classes da UML, por fim o projeto comportamental com diagrama de Atividades da SysML e diagrama de Requisitos da SysML.

Nenhum desses artigos relacionados mostrou todas essas características. Determinados artigos focam apenas na rastreabilidade de requisitos, outros na sincronização de modelos, e assim por diante (exceto para o trabalho apresentado em Riccobene e Scandurra (2012) cuja proposta é parecida com a desta dissertação, sendo que os autores do referido trabalho integram SysML e SystemC). Além disso, foram apresentadas linguagens com a mesma semântica, este aspecto é importante para facilitar o processo de aprendizagem.

5.4 Contribuições do Capítulo

Quando diferentes *concerns* precisam ser modelados em linguagens diferentes, os modelos individuais precisam ser sincronizados, de modo que as mudanças em um modelo sejam propagadas e refletidas nos outros. Existem esforços para sincronização e automatização por meio de ferramentas, mas praticamente isto ainda é um processo manual (BEHERE; TÖRNGREN, 2017). Assim, as ADLs necessitam ser expressivas o suficiente para modelar os *concerns* dos *stakeholders*.

Na técnica proposta neste trabalho, SysML e UML foram combinadas para descrever a

arquitetura de um sistema embarcado, com o objetivo principal de considerar todas as vantagens que ambas as linguagens combinadas têm. Além disso, a técnica une os esforços dos *stakeholders* na geração, comunicação e nas visões dos modelos em fases diferenciadas do ciclo de vida de um projeto, mas os mesmos modelos são continuamente combinados para que todos os envolvidos possam continuar o trabalho e o processo de criação e desenvolvimento do projeto seja entendido por todos e tenha evoluído até o produto ser finalizado.

As vantagens do diagrama de Requisitos e a tabela de Requisitos da SysML são reconhecidas como úteis em atividades de engenharia de requisitos na literatura (SOARES; VRANCKEN, 2007), (BEHJATI et al., 2011), (MARQUES; SIEGERT; BRISOLARA, 2014) e para modelagem de requisitos de sistemas embarcados (KHAN; MALLET; RASHID, 2015). Observou-se que um requisito modelado pela SysML pode aparecer em outros diagramas da UML e da SysML. Além disso, a combinação dos diagramas mostra a relação do requisito com o projeto. O relacionamento entre requisitos pode melhorar a especificação de sistemas, pois podem ser usados para modelar, documentar e analisar requisitos. O diagrama de Requisitos da SysML suporta rastreabilidade entre requisitos e modelagem de outros tipos de requisitos, além dos funcionais.

Conhecendo as vantagens da linguagem de modelagem SysML na engenharia de requisitos, as características gerais da SysML foram reconhecidas e os conceitos da SysML foram explorados para modelar a arquitetura de um sistema embarcado de tempo real. A técnica proposta integra SysML com UML para incluir SysML com elementos de modelagem de software, como Classes da UML.

Neste capítulo, descreveu-se o modo como as duas linguagens foram combinadas para fornecer uma linguagem de modelagem comum, a fim de especificar sistemas embarcados em diferentes níveis de abstração. O sistema de *airbag* é considerado como um exemplo inspirado pela realidade. Em comparação com os outros itens de conforto do corpo, o sistema de controle de *airbag* foi escolhido porque tem mais interação com outros componentes do carro, tem requisitos mais definidos, atividades em tempo real e representa um item de segurança do carro. Além disso, não foi encontrada pesquisa relacionada à modelagem de arquitetura de *airbag* com SysML ou outra linguagem.

A técnica apresentada usa linguagens com a mesma semântica para ajudar os engenheiros e arquitetos de software a se comunicarem e desenvolverem soluções de sistema otimizadas. Foi observado que UML e SysML são conhecidas e geralmente preferidas na indústria de software porque elas reduzem os custos de treinamento, reduzem o tempo de aprendizagem e possuem suporte de ferramentas adequado.

6 Modelagem da Arquitetura de Sistemas com SysML *versus* AADL

As vantagens da linguagem SysML têm sido reconhecidas para a engenharia de requisitos. Estas vantagens foram evidenciadas e têm sido analisadas para a modelagem de arquitetura de sistemas (EVENSEN; WEISS, 2010), (SHIRAISHI, 2013). Os diagramas de Requisitos, Atividades, Blocos e Blocos Internos, e a tabela de Requisitos são úteis para a representação arquitetural, principalmente por causa das visões arquiteturais diferenciadas que elas podem propiciar aos *stakeholders*. Por isto, neste capítulo é apresentada a modelagem de faróis automáticos de um sistema automotivo em duas linguagens: SysML e ADDL. Para o desenvolvimento da representação arquitetural com duas linguagens diferentes, têm-se como objetivos analisar os pontos fracos e fortes de cada linguagem, oferecer um exemplo comparativo e propor a SysML como uma linguagem de descrição arquitetural. Além disso, propor melhorias na SysML para a modelagem arquitetural. Vale salientar que a UML é atualmente a linguagem mais usada na indústria de software e diversas ferramentas para a modelagem baseada em UML estão disponíveis (SHIRAISHI, 2013).

A comunidade UML vem trabalhando com o intuito de apresentar uma forma de modelar as propriedades de sistemas. Dois desses esforços relacionados aos sistemas embarcados são os *profiles* SysML e MARTE. SysML permite capturar as interações com o mundo físico em um modelo matemático e a verificação de propriedades do modelo. Além disso, a SysML fornece um diagrama de Requisitos que suporta a descrição e a rastreabilidade dos requisitos (NIZ, 2007).

Neste capítulo, o foco principal é mostrar que a linguagem SysML pode ser usada para modelagem de arquitetura de sistemas. A comparação entre as linguagens SysML e AADL foi realizada para compreender as limitações que SysML tem e propor melhorias à SysML na descrição arquitetural.

6.1 Trabalhos relacionados

Shiraishi (2013) faz um levantamento dentre 60 ADLs possíveis para a descrição de um sistema automotivo. Após análise foram escolhidas 3 linguagens para a descrição arquitetural, AADL, MARTE e SysML. Portanto, Shiraishi (2013) mostra uma comparação entre as linguagens AADL e SysML. Nele é descrito todo o processo de modelagem do sistema de controle de velocidade de um automóvel com ambas as linguagens. MARTE (*profile* da UML) foi utilizada de forma complementar a SysML com o objetivo de modelar o tempo de execução do sistema. Contudo, o autor considera a AADL como a melhor alternativa para a representação arquitetural baseado nos critérios de geração de código, verificação formal, modelagem de erros

e modelagem de variabilidade.

Um levantamento das linguagens de modelagem de sistemas de software em tempo real é conduzido no artigo Evensen e Weiss (2010). Para os autores, cada linguagem tem suas vantagens e desvantagens e, para descrever adequadamente a arquitetura do sistema de software em tempo real, é quase certa a necessidade do uso complementar de duas ou mais linguagens. No trabalho são explorados AADL, UML, SysML, e MARTE para compreender o valor que cada linguagem traz para engenharia de software orientada a modelos e determinar se é viável combinar aspectos das quatro linguagens de modelagem para obter uma cobertura mais completa em descrições arquiteturais.

6.2 Uma visão geral da AADL

A AADL inclui abstrações de componentes de software, hardware e sistema para especificar e analisar sistemas embarcados em tempo real e mapeia software em elementos computacionais de hardware (FEILER; GLUCH; HUDAK, 2006).

Um componente em AADL é caracterizado por sua identidade (um nome exclusivo) e é definido por meio de declaração de tipo e de implementação, interfaces possíveis com outros componentes, propriedades e subcomponentes e suas interações são os principais elementos de especificação da AADL. Além de interfaces e elementos estruturais internos, outras abstrações podem ser definidas para uma arquitetura de componentes e sistemas. Por exemplo, informações a respeito de fluxo de dados ou controle de dados podem ser identificados, associados a componentes específicos e interligações. As abstrações de componentes da AADL são separadas em três categorias, são elas:

1. Software

    a) *Thread*: componente ativo que pode ser executado simultaneamente e organizado em grupos de *threads*;

    b) Grupo de *threads*: abstração de componente para organizar logicamente *thread*, dados e componentes de grupo de *threads* em um processo;

    c) Processo: espaço de endereço protegido cujos limites são aplicados em tempo de execução;

    d) Dados: tipos de dados e dados estáticos no código-fonte;

    e) Subprograma: conceitos como *call-return* e métodos de chamada (modelado usando um componente de subprograma que representa um pedaço de chamada de código fonte).

2. Plataforma de execução (hardware)

a) Processador: programa e executa *threads*;

b) Memória: armazena código e dados;

c) Dispositivo: representa sensores, atuadores ou outros componentes que interagem com o ambiente externo;

d) Barramento: interconecta processadores, memória e dispositivos.

3. Composto (*Composite*)

a) Sistema: elementos de design que permitem a integração de outros componentes em unidades distintas dentro da arquitetura.

Os componentes do sistema são combinados, ou seja, podem consistir de outros sistemas, bem como de componentes de software ou hardware. Os componentes são agrupados em categorias de software, plataforma de execução e categorias compostas (Ver Figura 6.1).

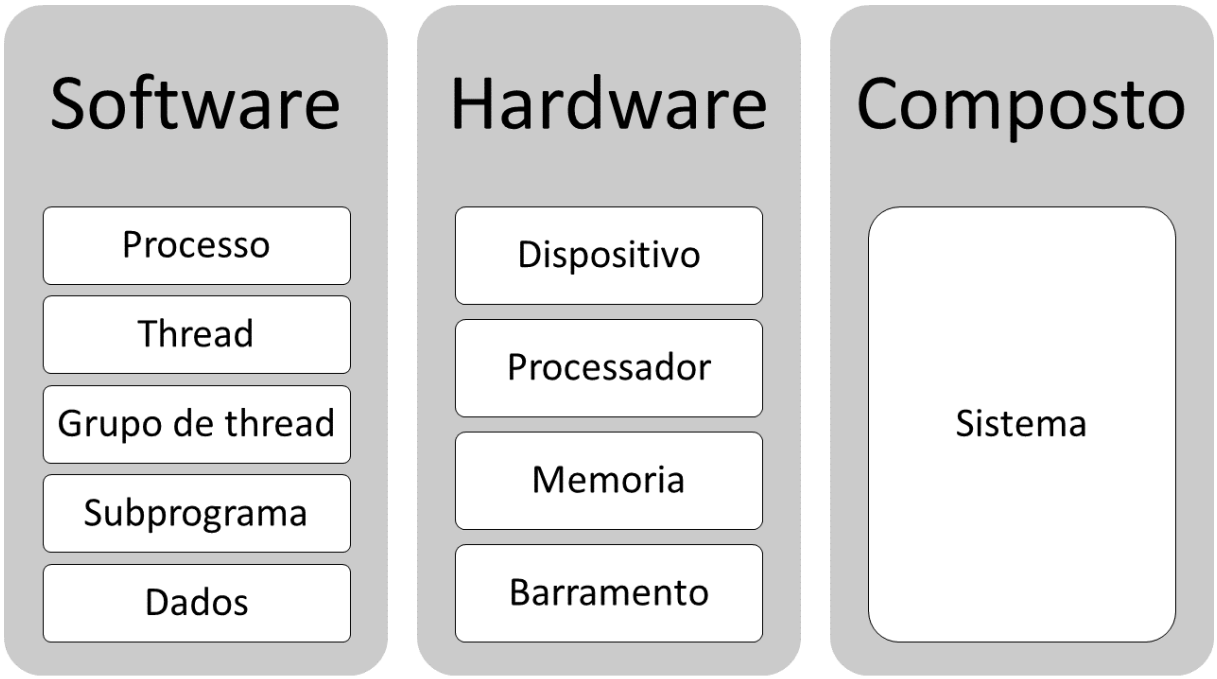

Figura 6.1 – Visão geral dos componentes de AADL.

Os diagramas da AADL apresentados neste trabalho foram feitos usando a ferramenta OSATE versão 2.2.1. OSATE é uma ferramenta *open-source* que suporta a linguagem AADL v2.2, fornecendo para AADL um editor textual, gráfico e XML, além de um conjunto de ferramentas de análise.

Os diagramas AADL são discutidos brevemente a seguir.

6.2.1 Especificação externa do sistema de faróis em AADL

As descrições em AADL consistem de elementos básicos chamados componentes. A modelagem da interface (Especificação externa) de componentes é dada pela definição de tipo

em AADL. Uma declaração de tipo de componente define os elementos de interface de um componente e atributos observáveis externamente, tais como dispositivos que fazem parte da interação com outros componentes e conexões (Ver Figura 6.2).

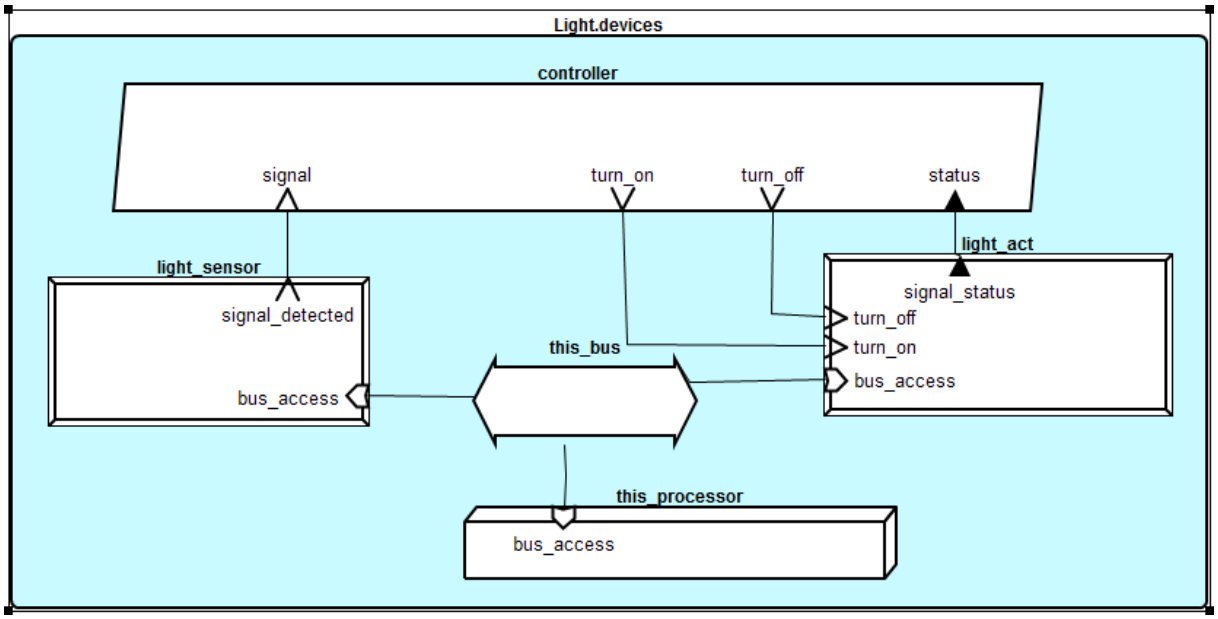

Figura 6.2 – Especificação externa do sistema de controle de faróis em AADL.

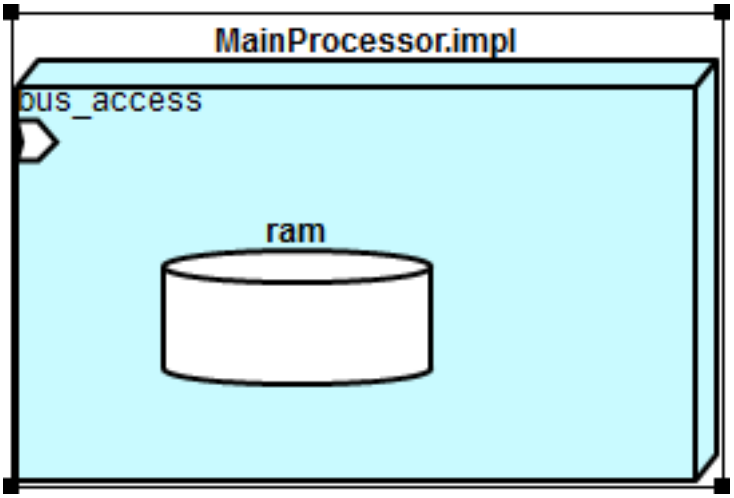

Figura 6.3 – Memória do sistema dos faróis modelados com AADL.

O sistema de controle de faróis na Figura 6.2 foi descrito na linguagem AADL. O sistema consiste de uma unidade de sistema, que contém um processador, um processo e dois dispositivos. A unidade de sistema recebe sinais de entrada vindos de um dispositivo de sensor nomeado por *"light_sensor"*. Quando a ausência ou presença de luz é detectada, o *"light_sensor"* envia o sinal, por meio de uma porta ao processo chamado *"controller"*. O processo recebe o sinal e o sistema começa a executar com a informação transmitida pelo sensor. Com o objetivo de executar o software, foi utilizado um processador e uma memória. A memória é um subcomponente do processador e está sendo representada na Figura 6.3. A memória é onde o código é armazenado

enquanto o processador está em execução. O barramento *"this_bus"* é requerido para carregar dados, controlar sinais e estabelecer comunicação apenas entre os componentes físicos, tais como dispositivos, processador e memória. As portas são usadas para estabelecer troca de dados e eventos entre componentes de hardware e software, como na comunicação entre o processo e o sensor.

6.2.2 Especificação interna do sistema de faróis em AADL

Uma declaração de implementação de componentes define a estrutura interna de um componente em termos de subcomponentes, conexões de subcomponentes, sequências de chamadas de subprograma, modos, implementações de fluxo e propriedades. Geralmente, a especificação externa e interna são realizadas repetindo duas etapas, alternadamente. No trecho de código na Figura 6.4 é apresentada a definição de tipo e a implementação de um processo em AADL. O processo nomeado por *"comm_controler"* e a implementação do processo contém e controla os *threads*.

```
--process
process comm controller
    features
        signal: in event port;
        status: in data port;
        turn_off: out event port;
        turn_on: out event port;
end comm_controller;

process implementation comm controller.threads
    subcomponents
        thread_readSignal: thread readSignalSensor.impl;
        thread_sendSignal: thread sendSignalAct.impl;
        thread_controlLight: thread controlLight.impl;
    connections
        Readsignal_thread_conn: port signal -> thread_readSignal.signal_in;
        Sendsignal_thread_conn: port status -> thread_sendSignal.status_in  ;
        controlLight_thread_conn: port thread_controlLight.turn_off_out-> turn_off;
        controlLightOn_thread_conn: port thread_controlLight.turn_on_out -> turn_on;
end comm_controller.threads;
```

Figura 6.4 – Código do processo em AADL.

No sistema de controle de faróis, os *threads* são subcomponentes do processo, ou seja, é uma especificação interna do processo, eles são apresentados na Figura 6.5. Estes *threads* devem ser capazes de receber como entrada o valor da intensidade de luz externa vindas do sensor (*"thread_readSignal"*), processar o sinal e transmitir ao atuador a resposta quanto ao acendimento automático dos faróis (*"thread_controlLight"*). Estes *threads* são conectados com portas de eventos, pois este tipo de porta permite o enfileiramento dos dados associados a um evento. O ultimo thread deve receber como resposta do atuador o estado atual dos faróis, se aceso ou apagado (*"thread_sendSignal"*). A memória apresentada na Figura 6.3 também consiste em uma especificação interna do hardware/processador.

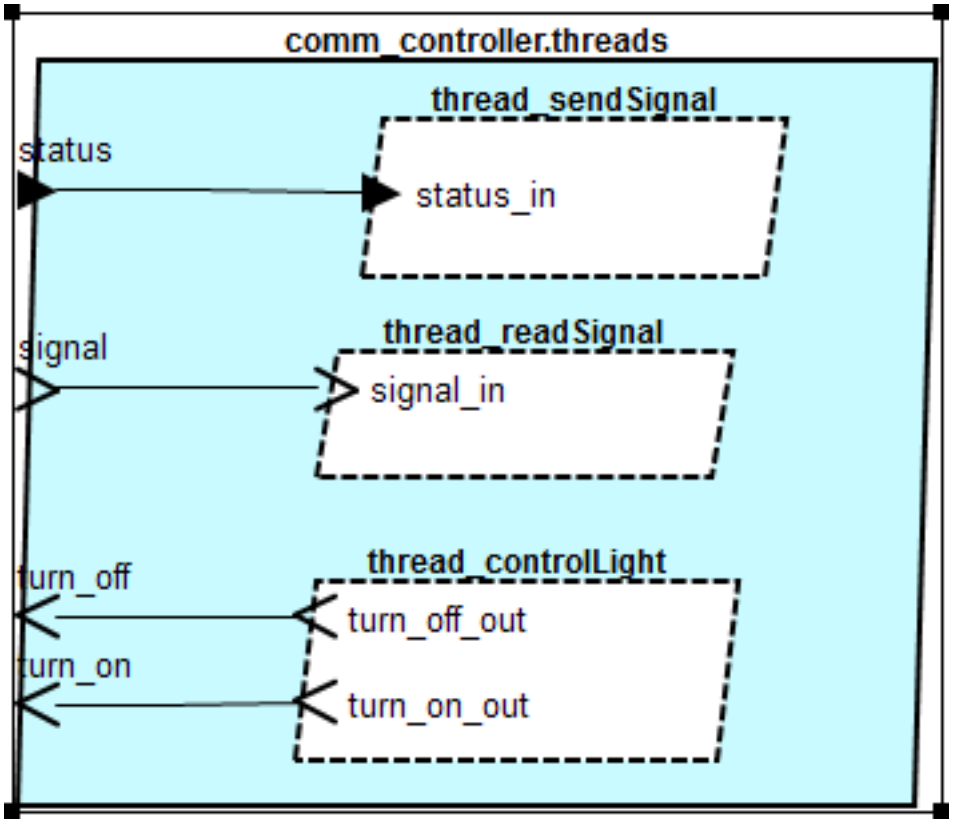

Figura 6.5 – Especificação interna do sistema de controle de faróis em AADL.

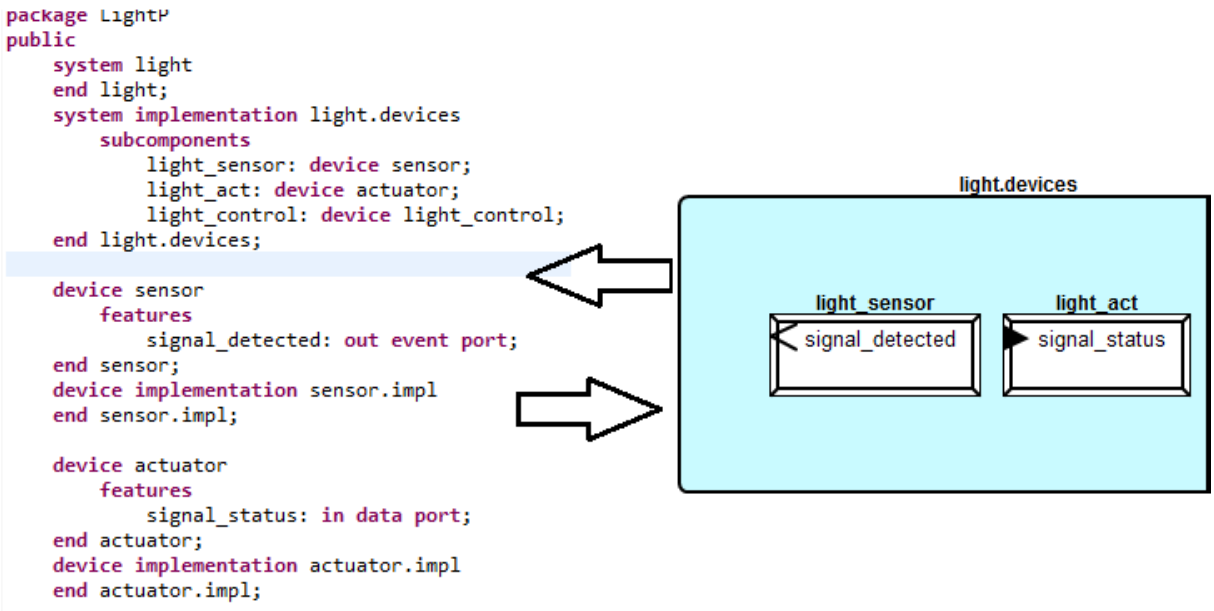

Figura 6.6 – Correlação entre a descrição textual e gráfica de AADL.

Um diferencial da linguagem AADL é que além dela permitir tanto a descrição textual, quanto gráfica do sistema, ela relaciona os dois. Assim, enquanto o projeto está sendo descrito textualmente pelo desenvolvedor, o modelo gráfico dele está sendo feito automaticamente pela ferramenta Osate. A descrição textual e gráfica podem ser visualizadas na Figura 6.6.

6.3 SysML

Nesta seção, o foco é no detalhamento da linguagem AADL e na comparação da AADL com a SysML.

Os diagramas da SysML apresentados neste trabalho foram feitos usando a versão Neon da ferramenta Eclipse, o plugin Papyrus 2.0.0 e a SysML 1.4 V. Os diagramas são mostrados a seguir.

6.3.1 Requisitos do sistema de faróis (diagrama e tabela)

O diagrama de Requisitos da SysML permite que cada requisito tenha um identificador (ID) e uma informação textual usando linguagem natural para descrever o requisito.

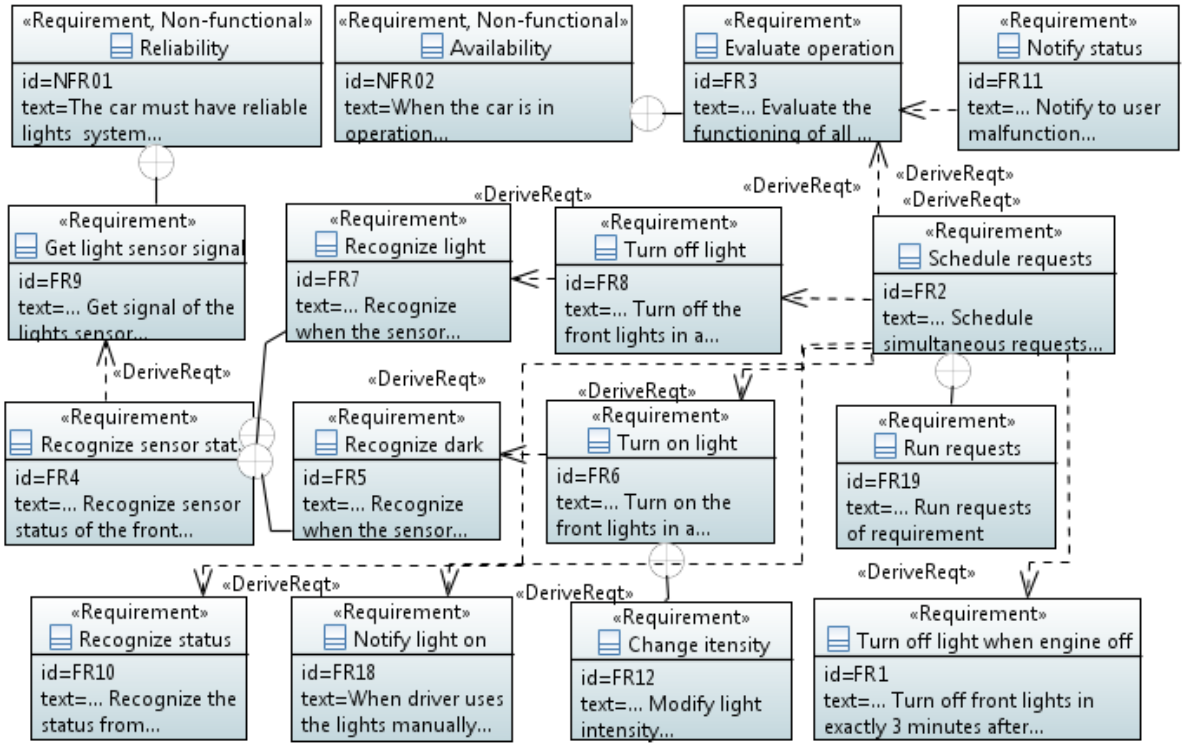

Figura 6.7 – Diagrama de requisitos (SysML) do sistema de faróis.

O diagrama de Requisitos de controle de faróis pode ser visto na Figura 6.7. O diagrama inclui uma descrição reduzida, pois devido a limitação de espaço, a descrição foi omitida. (Na Seção 3.3, há a descrição textual completa dos requisitos).

SysML também permite a representação de requisitos, suas propriedades e relacionamentos em um formato tabular que pode ser visto na Tabela 6.1.

Tabela 6.1 – Tabela de requisitos (SysML) do sistema de faróis.

id	name	type	derived Req	derivedFrom Req	containment Req
FR8	Turn off light	functional	FR2	FR7	
FR10	Recognize status	functional	FR2		
FR11	Notify status	functional		FR3	
NFR01	Reliability	non-functional			FR9
NFR02	Availability	non-functional			FR3
FR1	Turn off light when engine off	functional	FR2		
FR2	Schedule requests	functional		FR1, FR3, FR6, FR8, FR10, FR18	FR19
FR6	Turn on light	functional	FR2	FR5	FR12
FR4	Recognize sensor status	functional		FR9	FR5, FR7
FR5	Recognize dark	functional	FR6		
FR3	Evaluate operation	functional	FR2, FR11		
FR18	Notify light on	functional	FR2		
FR7	Recognize light	functional	FR8		
FR9	Get light sensor signal	functional	FR4		
FR12	Change itensify	functional			
FR19	Run requests	functional			

6.3.2 Diagrama de Definição de Bloco do sistema de faróis

A SysML fornece dois diagramas típicos para modelagem de arquitetura de sistema: BDD e IBD. Estes dois diagramas são semelhantes à definição de tipo e implementação de AADL, respectivamente.

No exemplo, o BDD foi desenvolvido para modelar componentes envolvidos no sistema de faróis. As funções especificadas são implementadas para comunicações seguras com outros componentes e com sensores/atuadores locais. O BDD da SysML é representado na Fig. 6.8.

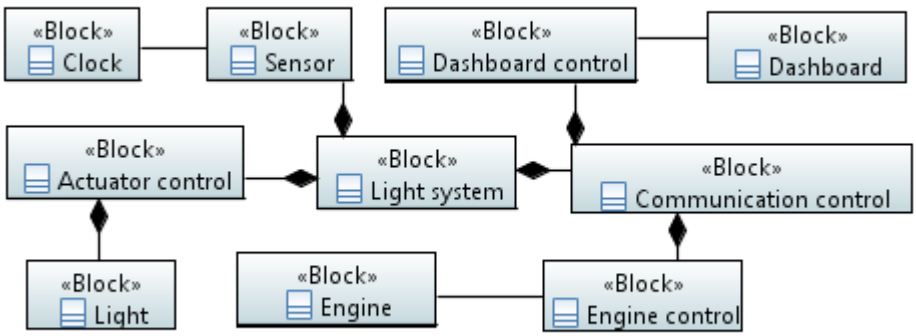

Figura 6.8 – Diagrama BDD (SysML) do sistema de faróis.

6.3.3 Diagrama de Bloco Interno do sistema de faróis

Foi utilizado um IBD para a modelagem das operações internas do bloco *Light System*, este diagrama é mostrado na Figura 6.9.

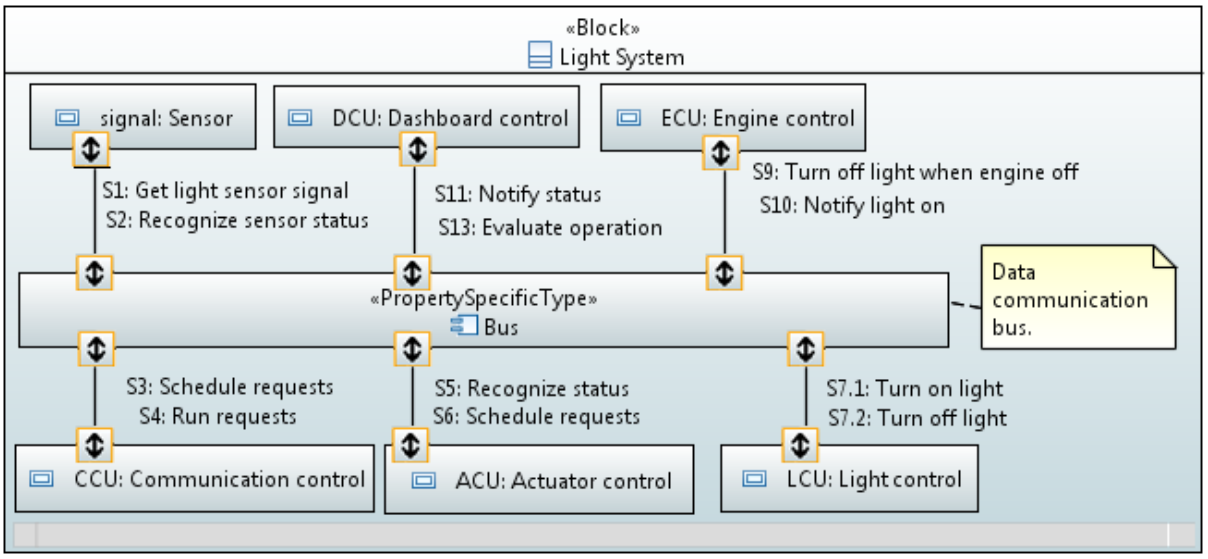

Figura 6.9 – Diagrama IBD (SysML) do sistema de faróis.

6.3.4 Diagrama de atividades SysML

A atividade principal dos faróis é representada na Fig. 6.10. Foram modeladas as atividades inicial e final, nó de decisões que representa a ação de ligar ou desligar os faróis, laço iterativo com ⟨⟨loop node⟩⟩ e restrições com OCL para o a repetição do *loop* e para desligar os faróis em exatos 3 minutos após o carro ser desligado.

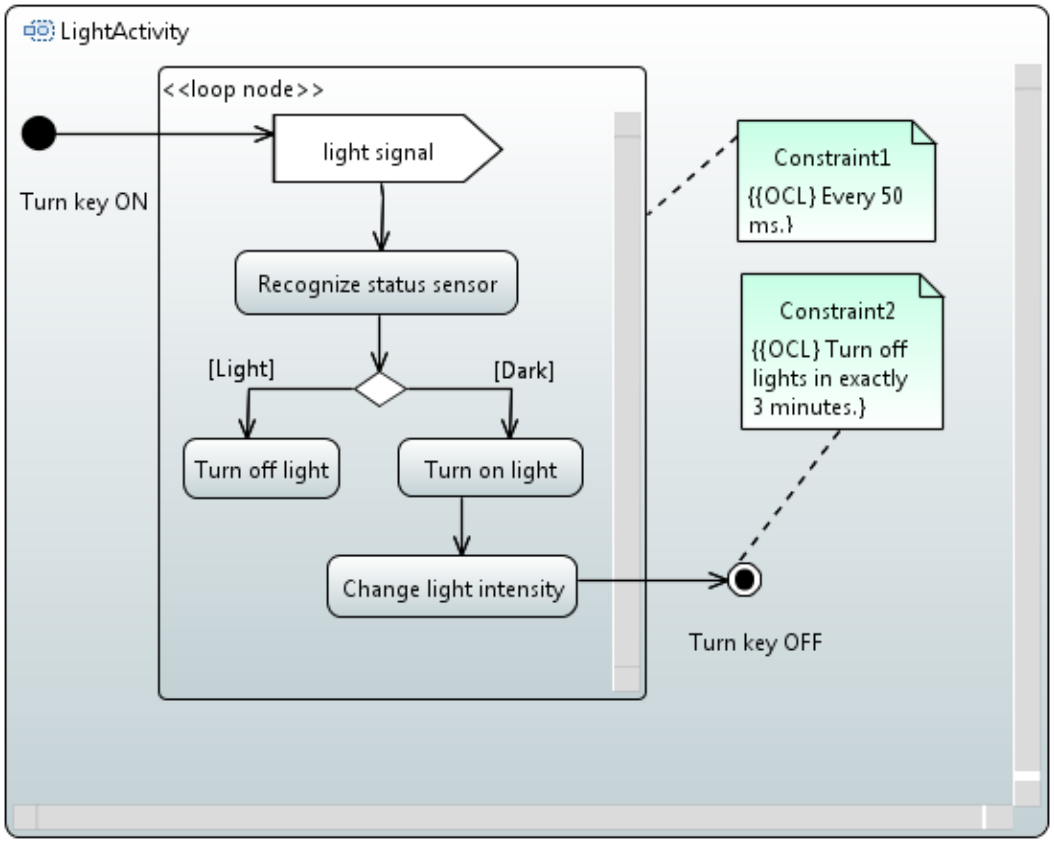

Figura 6.10 – Diagrama de atividades (SysML) do sistema de faróis.

6.4 Uma Análise Comparativa entre SysML e AADL

Em termos práticos, percebeu-se que características abstratas, tais como tomadas de decisão, repetição de uma funcionalidade (*loop*), representação do início e fim de uma atividade, características que são relacionadas à realidade e, consequentemente, ao sistema, não podem ser descritas em AADL. A falta destas características foi observada enquanto os faróis estavam sendo modelados em AADL.

Não há, em AADL, um diagrama específico para representar requisitos e outro para mostrar o comportamento por meio de ações, como o diagrama de Atividades da SysML. Além disso, não é possível mostrar a relação de requisitos com o projeto de sistemas com AADL e também não se pode mostrar a relação de projeto de sistema com o projeto de software.

Os relacionamentos em AADL não são diferenciados. Quando são relacionados dois tipos de sistemas, apenas pode-se relacioná-los como "extensão", isto é, um sistema que estende outro. Não existe outro tipo de classificação para relacionamentos como há em SysML, por exemplo, *Trace, Derive, ou Containment*. Não existe a possibilidade de relacionar um tipo de sistema a um dispositivo, por exemplo.

Na AADL, o componente é identificado somente pelo nome, ele não tem um ID. O ID é essencial para identificar e rastrear um requisito facilmente, por exemplo, como mostrado na Tabela 6.1.

O desenvolvimento de diagramas na linguagem gráfica de AADL é complexo, é necessário alterar o diagrama interno para que o relacionamento seja alterado no diagrama externo. Pode acontecer que esta manipulação do diagrama interno para o externo ou do externo para o interno tornar-se confusa por causa das portas e conexões, considerando que as conexões são difíceis de implementar no modelo gráfico. Contudo, a linguagem textual torna a AADL mais fácil de manipular, o exemplo do sistema de controle de faróis foi feito exclusivamente por código textual.

No caso da linguagem gráfica da SysML, usando a ferramenta Eclipse, os diagramas podem ser sincronizados e desenvolvidos simultaneamente, por exemplo, se o desenvolvedor acrescenta um bloco ao sistema no gráfico BDD, o mesmo bloco é gerado automaticamente no IBD.

A SysML fornece diagramas para a modelagem de comportamento, como o diagrama de Sequência, o diagrama de Atividades (Figura 6.10) e o diagrama de Máquina de Estados. SysML tem forte capacidade de modelagem de comportamento do sistema. Já a AADL fornece um limitado vocabulário para modelagem de comportamento (SHIRAISHI, 2013).

6.5 Proposta de melhorias na SysML para Modelagem de Arquitetura

A AADL apresenta a implementação do modelo por meio de uma descrição textual. A SysML tem um editor textual para editar configurações referenciáveis a cada elemento existente no modelo. A primeira proposta que é necessária para a SysML é conduzir a implementação do modelo por meio de descrição textual, que possa efetivamente definir os componentes da SysML em sintaxe textual e mapeá-la para sintaxe gráfica.

A segunda proposta para SysML consiste em considerar um padrão para descrição de componentes tais como memória, barramento ou processo. Todos os conceitos de componentes são descritos como blocos, ou seja, tudo é modelado independentemente de classificação (isso foi observado em BDD, IBD, e no diagrama de Atividades, que possui pequenas alterações de classificações diferenciadas, tais como eventos e sinais) diferentemente de AADL que tem definido todos os conceitos de forma diferenciada. Dessa forma, é proposta uma representação entre os componentes do sistema para os diagramas BDD e IBD da SysML, principalmente a diferença clara entre os componentes de hardware e software existentes, a fim de esclarecer detalhadamente cada componente descrito no projeto.

Shiraishi (2013) afirma que os engenheiros que trabalham na fabricação de automóveis geralmente precisam ter um ponto de vista superior com visão para o sistema geral (todo/completo) do veículo, a linguagem UML não oferece este tipo de suporte (SHIRAISHI, 2013). Em outras palavras, a arquitetura do sistema global é essencial para que os engenheiros automotivos visualizem os sistemas completamente, em vez dos detalhes de cada sistema (SHIRAISHI, 2013). Consequentemente, uma das propostas de melhorias para SysML consiste na técnica apresentada no Capítulo 5, com ela tem-se a possibilidade de documentação completa dos sistemas, pois a técnica oferece suporte a modelos integrados de software, sistemas e requisitos, possibilitando três visões arquiteturais por meio da junção de diagramas da UML e da SysML: a visão estrutural (Fig. 5.6), a visão comportamental (Fig. 5.8) e a visão de Requisitos (Fig. 5.3).

A maioria dos sistemas automotivos são sistemas de tempo real. A última melhoria proposta é incorporar propriedades de tempo real na SysML, como propriedades de tempo modelados junto a arquitetura. Nos exemplos anteriores, as propriedades de tempo foram descritas de forma textual na Seção 3.3, quando os requisitos eram descritos de acordo com a Norma ISO 29148:2011 e como restrições na linguagem OCL no diagrama de Atividades da Fig. 6.10.

6.6 Contribuições do Capítulo

Foi observado que a SysML pode ser utilizada amplamente para modelagem de características abstratas. SysML captura precisamente os requisitos e o gerenciamento da complexidade do sistema é tratado desde as etapas iniciais de desenvolvimento de um sistema com a expressividade

do diagrama de Requisitos.

A técnica proposta permite visualização e reconhecimento de quais requisitos se comunicam com outros, por meio do diagrama e da tabela de Requisitos, e reconhece qual requisito interage com outros sistemas (blocos). Consequentemente, identificar qual subsistema pode afetar os requisitos não funcionais, tais como o desempenho, ou segurança no sistema principal.

O diagrama de Blocos e diagrama de Blocos Internos capturam os sistemas envolvidos no projeto, propiciam a comunicação entre eles e os recursos necessários. O diagrama de Atividades pode representar os passos necessários para uma determinada atividade ser concluída com sucesso, ou pode mostrar um desvio previsto. Este diagrama também captura as principais decisões estratégicas de um sistema, sendo também útil para a comunicação entre os *stakeholders*.

Pode-se concluir que os diagramas de Requisitos, Atividades, Blocos e Blocos Internos são necessários para a modelagem de arquitetura de software. Porém, a linguagem SysML precisa ser ampliada na classificação de blocos, implementação do modelo por meio da sintaxe textual e descrição de propriedades de tempo para possuir características necessárias para modelagem da arquitetura de sistemas automotivos de tempo real.

7 Conclusão

No Capítulo 1 foi evidenciada a evolução e a complexidade dos sistemas de software. A arquitetura de software tem sido estudada para mostrar que a abstração vem ajudando os *stakeholders* na condução e entendimento de todo o processo de construção de um projeto. Mediante uma arquitetura bem planejada e desenvolvida, a descrição do sistema corrobora com a documentação, comunicação entre os envolvidos e entendimento de todas as partes. Porém, as diversas linguagens de descrição de arquitetura de software não vêm acompanhando o formato dos novos sistemas.

Os sistemas de tempo real, por exemplo, têm sido estudados intensivamente, uma vez que a sociedade tem se tornado dependente destes sistemas. Para o domínio de sistemas embarcados automotivos, que exige um alto nível de confiabilidade, são necessárias descrições dos requisitos do sistema, projeto arquitetural bem desenvolvido, planejamento de toda a estrutura tanto em termos de software quanto de hardware, linguagem utilizada e ferramenta para lidar com os complexos problemas que surgem relacionados a tempo e segurança. Neste contexto, diversas linguagens de modelagem foram propostas.

As informações a respeito de linguagens arquiteturais e sistemas embarcados de tempo real são descritas no Capítulo 2.

No Capítulo 3 foram apresentados os requisitos de acordo com a Norma ISO 29148 para o sistema de controle de *airbag* e para o sistema de controle de faróis, ambos os sistemas foram utilizados para a construção de dois exemplos inspirados na realidade, o primeiro para a construção de uma técnica que amplia a visão arquitetural de sistema e correlaciona a arquitetura de sistema com a arquitetura de software, e o segundo exemplo para realizar a comparação entre a AADL e SysML.

No Capítulo 4 foram consultadas, analisadas e comparadas dez linguagens em relação a dezesseis características de arquitetura de software. Esta comparação serviu para delinear a pesquisa a respeito das ADLs e evidenciar quais ADLs eram capazes de descrever sistemas complexos, de acordo com as novas características incorporadas no projeto de um software e que foram levantadas nesta pesquisa. Foi considerado, como um dos resultados desta análise, que a maioria das ADLs não suporta múltiplas visões arquiteturais, o suporte a múltiplas visões arquiteturais é conhecido como uma boa prática na arquitetura de software.

A proposta elaborada no Capítulo 5 descreve a combinação de diagrama de requisitos com BDD da SysML, diagrama de Atividades com diagrama de Requisitos da SysML e diagrama de Classes da UML com o BDD da SysML. Além disso, a técnica descreve a combinação da tabela de Requisitos com novos campos, mostrando o relacionamento de requisitos e conexões ao bloco associado a um determinado requisito, contribuindo para o rastreamento de requisitos

em diferentes níveis de abstração. A introdução desta nova técnica fornece uma ligação entre representações de requisitos, software e sistema em diferentes níveis de abstração, fundamenta a representação das relações de rastreabilidade e composição entre requisitos modelados e integra as vantagens de UML e SysML para incluir SysML com elementos de modelagem de software.

A técnica proposta no Capítulo 5 foi aplicada por meio de estudo de caso em um conjunto de requisitos para um sistema de controle de *airbag*. O sistema de *airbag* é um sistema crítico de software de tempo real, em que diversos elementos estão envolvidos, e são aplicados a uma infraestrutura crítica. O projeto desse sistema aborda critérios de desempenho, confiabilidade e segurança, uma vez que vidas humanas estão diretamente envolvidas e que a infraestrutura veicular é fundamental para a segurança. O uso de descrições gráficas no processo de desenvolvimento de sistemas embarcados de tempo real permite representar o sistema sob diferentes perspectivas possibilitando na especificação do mesmo descrever os requisitos, componentes de hardware e software (como blocos), comportamento e estrutura de um sistema.

As linguagens SysML e AADL foram comparadas no Capítulo 6 para mostrar as vantagens da SysML e propor melhorias na representação consistente relativos a sistemas de tempo real. Evidenciou-se também que o uso de linguagens gráficas aumenta a inteligibilidade do sistema, uma vez que permite aos projetistas fornecer descrições de alto nível do sistema.

A arquitetura de sistemas de tempo real é dependente do acoplamento entre a função do sistema e o tempo. Portanto, conclui-se que os conceitos de SysML e UML, articulados por meio da combinação de diagramas são complementares. A técnica proposta da junção de UML e SysML fornece elementos para descrever requisitos de software e seus relacionamentos com o sistema, gerenciar mudanças, evoluir e rastrear requisitos mais facilmente, além da comunicação ser efetivamente realizada entre os *stakeholders*. Este aspecto é importante ao desenvolvimento de sistemas de tempo real, por causa da diversidade de pessoas/equipes envolvidas e que influenciam uma ampla série de decisões de projeto.

Neste trabalho, estratégias que contribuem para a arquitetura de software que visam identificar requisitos, descrever a arquitetura e relacionar projetos arquiteturais de software e de sistemas de tempo real foram elaboradas.

A SysML foi projetada de forma a modelar sistemas em diversos domínios, ou seja, ela não é uma linguagem com foco restrito, como a AADL. A SysML possibilita a modelagem de componentes de software, hardware e sistemas, suporta tipos diferentes de conectores, tais como relacionamentos hierárquicos e derivados. Permite ainda a modelagem em múltiplas visões arquiteturais, por meio dos diagramas que podem modelar a estrutura, o comportamento e os requisitos do sistema.

Existem ferramentas de desenvolvimento para modelar arquiteturas de software com SysML, dentre elas pode-se destacar: *Agilian, Artisan Studio, Enterprise Architect, Cameo*

Systems Modeler, Rhapsody, UModel. Modelio e Papyrus são ferramentas de modelagem *open-source*. *Papyrus* é uma ferramenta gráfica de modelagem UML 2 com base no ambiente Eclipse, ela fornece geração de código (C, C ++, Java) e vem com um conjunto de plugins de extensão pré definidos dedicados a aplicações embarcadas de tempo real, incluindo padrões da SysML.

Na linguagem AADL, a descrição do comportamento do sistema é limitada e o suporte a conectores também. Não é possível descrever os requisitos do sistema, mas os requisitos não funcionais podem ser especificados no diagrama interno do sistema. Por exemplo, os componentes de *thread* suportados por AADL incluem as opções de propriedades de execução pré declaradas de eventos periódicos, aperiódicos (dirigido a eventos), *background* (disparado uma vez e executado até a conclusão) e eventos esporádicos (estimulado por um limite de taxa superior). Em AADL, o domínio é específico apenas para sistemas embarcados de tempo real. Somente há dois tipos de visões arquiteturais gráficas, a especificação interna e externa, e um tipo de visão textual. Em outras palavras, as visões são também limitadas.

7.1 Contribuições do Trabalho

Além das vantagens citadas e pertinentes à linguagem SysML, com a técnica sugerida no Capítulo 5 de integrar a UML e a SysML, é possível descrever os requisitos de usuários em alto nível, as diversas visões de sistema e de software combinadas, a estrutura em diferentes níveis de abstração e o comportamento dinâmico em diferentes níveis de abstração. É possível, também, especificar, classificar e rastrear a relação dos requisitos com modelos de sistemas dinâmicos (com o conector *"satisfy"*), a relação de requisitos com modelos de sistemas estruturais (com o conector *"satisfy"*) e a relação estrutural de sistemas com software (por meio de blocos e classes).

Podem ser citadas as principais contribuições para a comunidade científica quanto à realização deste trabalho:

- Foram especificados requisitos para o sistema de controle de *airbag* e para o sistema de controle de faróis de acordo com a Norma ISO/IEC/IEEE 29148:2011. Os dois sistemas são sistemas embarcados de tempo real. A especificação desses requisitos é uma contribuição deste trabalho, pois não foram encontradas na literatura referências quanto a identificação e especificação de requisitos para esses dois sistemas.

- Foi realizada uma análise comparativa com 10 ADLs usando 16 critérios especificados para a descrição arquitetural. Essa análise serviu para delinear a pesquisa, mostrando a situação atual das ADLs e sua evolução ao longo dos anos. Foram discutidos os progressos, problemas e considerações sobre a aplicação de ADLs na prática.

- Foi proposta uma técnica para modelar a arquitetura de software de tempo real usando em conjunto as linguagens de modelagem UML e SysML.

- Foram modeladas partes das arquiteturas de dois sistemas (faróis e *airbag*) usando a SysML. O sistema de controle de *airbag* foi modelado para apresentar uma técnica de modelagem arquitetural combinando os elementos de software (UML) com os elementos de sistema (SysML). A técnica viabiliza documentação de software utilizando apenas duas linguagens ao longo de todo o desenvolvimento de software, viabilizando também a comunicação entre os *stakeholders* e múltiplas visões arquiteturais. O sistema de controle de faróis foi modelado para comparação com uma ADL, a AADL.

- Foi modelada e apresentada a arquitetura do sistema de controle de faróis usando AADL com o objetivo de compará-la a SysML, mostrar as vantagens e desvantagens da linguagem.

- Foi feita a análise comparativa entre as linguagens SysML e AADL, além das vantagens e desvantagens apresentadas, foram propostas melhorias para que a SysML possua características necessárias para modelagem da arquitetura de sistemas automotivos de tempo real.

- Esta pesquisa também resultou em dois trabalhos em congressos científicos, são eles:

  – RIBEIRO, Q. A. D. S. SOARES, M. S. VRANCKEN, Jos *"A Comparative Analysis of Software Architecture Description Languages"* submetido à *17th International Conference on Computational Science and Its Applications* (ICCSA, 2017), evento qualificado com *Qualis* B1.

  – RIBEIRO, Q. A. D. S. RIBEIRO, F. G. C. SOARES, M. S. *"A Technique to Architect Real-Time Embedded Systems with SysML and UML through Multiple Views"* aceito na *19th International Conference on Enterprise Information Systems* (ICEIS, 2017), evento qualificado com *Qualis* B1.

7.2 Trabalhos Futuros

No domínio da arquitetura de sistemas de tempo real e embarcados, sugere-se como trabalhos futuros:

- Desenvolver um plugin para o Eclipse que ajudará na integração de UML e SysML para modelagem da arquitetura de sistemas.

- Desenvolver materiais de treinamento para a técnica proposta.

- Apresentar e avaliar esta técnica para arquitetos de sistemas para que eles apresentem melhorias para SysML (além daquelas melhorias que foram propostas neste trabalho).

Referências

ALLEN, R.; GARLAN, D. The WRIGHT Architectural Description Language. *Technical Report, Carnegie Mellon University*, 1996.

ALMARI, H.; BOUGHTON, C. The Five Factors Influencing Software Architecture Modeling and Evaluation Techniques. *International Conference on IT Convergence and Security (ICITCS)*, p. 1–8, 2014.

AUWERAER, H. Van der et al. Virtual Engineering at Work: the Challenges for Designing Mechatronic Products. *Engineering with Computers*, Springer, v. 29, n. 3, p. 389–408, 2013.

BASHROUSH, R. et al. Adlars: An Architecture Description Language for Software Product Lines. *29th Annual IEEE/NASA Software Engineering Workshop*, IEEE, p. 163–173, 2005.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 3ª. ed. United States of America: Addison-Wesley Professional, 2012.

BEHERE, S.; TÖRNGREN, M. Systems Engineering and Architecting for Intelligent Autonomous Systems. *Automated Driving*, Springer, p. 313–351, 2017.

BEHJATI, R. et al. Extending SysML with AADL Concepts for Comprehensive System Architecture Modeling. *European Conference on Modelling Foundations and Applications*, Springer, p. 236–252, 2011.

BLANC, X. et al. Detecting Model Inconsistency Through Operation-based Model Construction. *Proceedings of the 30th international conference on Software engineering*, ACM, p. 511–520, 2008.

BRAGA, I. N. C. *Como Funcionam os Sistemas Eletrônicos do Automóvel (ART567)*. 2016. <http://www.newtoncbraga.com.br/index.php/como-funciona/4142-art567>. Acesso em: 28 março 2017.

BROWN, A. W. Model Driven Architecture: Principles and Practice. *Software and Systems Modeling*, Springer, v. 3, n. 4, p. 314–327, 2004.

BUREŠ, T.; HNĚTYNKA, P.; PLÁŠI, F. Sofa 2.0: Balancing Advanced Features in a Hierarchical Component Model. *Fourth International Conference on Software Engineering Research, Management and Applications*, IEEE, p. 40–48, 2006.

BURES, T.; PLASIL, F. Communication Style Driven Connector Configurations. *Software Engineering Research and Applications*, p. 102–116, 2004.

CARROS-ONLINE. *Como Funcionam os Faróis Automáticos?* 2016. <http://www.carro-carros.com/carros/driving-safety/driving-safety/146571.html>. Acesso em: 29 março 2017.

ČERNÝ, O. et al. *SOFA 2 Component System: User's Guide*. Prague: Charles University in Prague, 2015. 1-147 p.

CHARETTE, R. N. This Car Runs on Code. *IEEE spectrum*, IEEE, v. 46, n. 3, p. 3, 2010.

CHEN, F. et al. An Architecture-based Approach for Component-Oriented Development. *Proceedings of the 26th Annual International Computer Software and Applications Conference*, IEEE, p. 450–455, 2002.

CHIAO, C. M.; KUNZLE, V.; REICHERT, M. Integrated Modeling of Process-and Data-centric Software Systems with PHILharmonicFlows. *IEEE 1st International Workshop on Communicating Business Process and Software Models Quality, Understandability, and Maintainability (CPSM)*, p. 1–10, 2013.

CLEMENTS, P. C. A Survey of Architecture Description Languages. *Proceedings of the 8th International Workshop on Software Specification and Design*, IEEE, p. 16, 1996.

DAI, L.; COOPER, K. Modeling and Analysis of Non-functional Requirements as Aspects in a UML Based Architecture Design. *First ACIS International Workshop on Self-Assembling Wireless Networks*, IEEE, p. 178–183, 2005.

DASHOFY, E. M.; HOEK, A. Van der; TAYLOR, R. N. A Highly-Extensible, XML-based Architecture Description Language. *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, IEEE, p. 103–112, 2001.

EVENSEN, K.; WEISS, K. A Comparison and Evaluation of Real-time Software Systems Modeling Languages. *AIAA Infotech@ Aerospace 2010*, American Institute of Aeronautics and Astronautics, p. 3504, 2010.

FEILER, P. H.; GLUCH, D. P.; HUDAK, J. J. *The Architecture Analysis & Design Language (AADL): An Introduction*. Pittsburgh PA: Carnegie Mellon University, 2006.

FEILER, P. H. et al. *System Architecture Virtual Integration: An Industrial Case Study*. Pittsburgh PA: Carnegie Mellon University, 2009.

FEILER, P. H.; LEWIS, B. A.; VESTAL, S. The SAE Architecture Analysis & Design Language (AADL) a Standard for Engineering Performance Critical Systems. *IEEE 2006 - Computer Aided Control System Design, IEEE 2006 - International Conference on Control Applications, IEEE 2006 - International Symposium on Intelligent Control*, IEEE, p. 1206–1211, 2006.

FRANCA, R. B. et al. The AADL Behaviour Annex–experiments and Roadmap. *12th IEEE International Conference on Engineering Complex Computer Systems*, IEEE, p. 377–382, 2007.

GARDAZI, S. U. et al. Survey of Software Architecture Description and Usage in Software Industry of Pakistan. *International Conference on Emerging Technologies*, IEEE, p. 395–402, 2009.

GARLAN, D.; KOMPANEK, A. J. Reconciling the Needs of Architectural Description with Object-modeling Notations. *UML 2000 - The Unified Modeling Language*, Springer, p. 498–512, 2000.

GARLAN, D.; SHAW, M. *An Introduction to Software Architecture*. Pittsburgh, PA, USA, 1994.

GIESE, H.; HILDEBRANDT, S.; NEUMANN, S. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. *Graph transformations and model-driven engineering*, Springer-Verlag, p. 555–579, 2010.

GOMAA, H.; WIJESEKERA, D. *The Role of UML, OCL and ADLs in Software Architecture*. Toronto, Canada: Proc. Workshop Describing Software Architecture with UML, ICSE'01, 2001.

GOULÃO, M.; ABREU, F. B. e. Bridging the gap Between Acme and UML 2.0 for CBD. *Proceedings of Specification and Verification of Component-Based Systems (SAVSCB'03), workshop at ESEC/FSE 2003*, p. 75–79, 2003.

GROUP, O. M. et al. A UML profile for MARTE: Modeling and Analysis of Real-time Embedded Systems, Beta 2 (Convenience Document with Change Bars). *OMG MARTE documentation*, p. 1–676, 2008.

HAUSE, M. et al. The SysML modelling language. *Fifteenth European Systems Engineering Conference*, v. 9, 2006.

HILLIARD, R. IEEE-std-1471-2000 Recommended Practice for Architectural Description of Software-intensive Systems. *IEEE, http://standards. ieee. org*, v. 12, p. 16–20, 2000.

HILLIARD, R. et al. On the Composition and Reuse of Viewpoints Across Architecture Frameworks. *Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA)*, IEEE, p. 131–140, 2012.

HILLIARD, R.; RICE, T. Expressiveness in Architecture Description Languages. *Proceedings of the third international workshop on Software architecture*, p. 65–68, 1998.

HNETYNKA, P.; BURES, T. Advanced Features of Hierarchical Component Models. *Proceedings of ISIM, Hradec nad Moravici*, p. 3–10, 2007.

ISO, I. IEC 15288: 2008. Systems and Software Engineering - System Life Cycle Processes. *2nd International Organization for Standardization/International Electrotechnical Commission*, ISO/IEEE/IEC, 2008.

ISO, I. IEEE. 29148: 2011- Systems and Software Engineering-Requirements Engineering. ISO/IEEE/IEC, 2011.

ISO:IEC:IEEE:42010. Systems and Software Engineering: Architecture Description. ISO/IEC/IEEE Standard, 2011.

KANDÉ, M. M. et al. Bridging the Gap Between IEEE 1471, an Architecture Description Language, and UML. *Software and Systems Modeling*, Springer, v. 1, n. 2, p. 113–129, 2002.

KHAN, A. M.; MALLET, F.; RASHID, M. Modeling SystemVerilog Assertions using SysML and CCSL. *Electronic System Level Synthesis Conference*, ESLsyn Conference, 2015.

KOOPMAN, P. *Better Embedded System Software*. Carnegie Mellon University: Drumnadrochit Education, 2010.

KRUCHTEN, P. What Do Software Architects Really Do? *Journal of Systems and Software*, v. 81, n. 12, p. 2413–2416, 2008.

KRUCHTEN, P.; STAFFORD, J. The Past, Present, and Future for Software Architecture. *IEEE Software*, v. 23, n. 2, p. 22–30, 2006.

KRUCHTEN, P. B. The 4+ 1 View Model of Architecture. *IEEE Software*, v. 12, n. 6, p. 42–50, 1995.

LANGE, C. F.; CHAUDRON, M. R.; MUSKENS, J. In Practice: UML Software Architecture and Design Description. *IEEE Software*, v. 23, n. 2, p. 40–46, 2006.

LI, Q.; YAO, C. Real-time Concepts for Embedded Systems. CRC Press, 2003.

LINHARES, M. V. et al. Introducing the Modeling and Verification Process in SysML. *Conference on Emerging Technologies and Factory Automation, ETFA*, IEEE, p. 344–351, 2007.

LUCAS, F. J.; MOLINA, F.; TOVAL, A. A Systematic Review of UML Model Consistency Management. *Information and Software Technology*, Elsevier, v. 51, n. 12, p. 1631–1645, 2009.

LUCKHAM, D. C. *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events*. Stanford, CA, USA, 1996.

MAGEE, J.; DULAY, N.; KRAMER, J. Structuring Parallel and Distributed Programs. *Software Engineering Journal*, v. 8, n. 2, p. 73–82, 1993.

MAGEE, J.; DULAY, N.; KRAMER, J. Regis: A Constructive Development Environment for Distributed Programs. *Distributed Systems Engineering*, v. 1, n. 5, p. 304–312, 1994.

MAGEE, J.; KRAMER, J. Dynamic Structure in Software Architectures. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 21, n. 6, p. 3–14, 1996.

MALAVOLTA, I. et al. What Industry needs from Architectural Languages: A Survey. *IEEE Transactions on Software Engineering*, v. 39, n. 6, p. 869–891, 2013.

MARQUES, M. R. S.; SIEGERT, E.; BRISOLARA, L. Integrating UML, MARTE and SysML to Improve Requirements Specification and Traceability in the Embedded Domain. *12th IEEE International Conference on Industrial Informatics (INDIN)*, IEEE, p. 176–181, 2014.

MARWEDEL, P.; GOOSSENS, G. *Code Generation for Embedded Processors*. [S.l.]: Springer Science & Business Media, 2013. v. 317.

MEDVIDOVIC, N. Formal Definition of the Chiron-2 Software Architectural Style. Department of Information and Computer Science, University of California, Irvine, 1995.

MEDVIDOVIC, N. *A Classification and Comparison Framework for Software Architecture Description Languages*. Irvine, California: Department of Information and Computer Science, University of California, 1996. 1 – 50 p.

MEDVIDOVIC, N.; DASHOFY, E. M.; TAYLOR, R. N. Moving Architectural Description from under the Technology Lamppost. *Information and Software Technology*, v. 49, n. 1, p. 12–31, 2007.

MEDVIDOVIC, N.; EGYED, A.; ROSENBLUM, D. S. Round-trip Software Engineering Using UML: From Architecture to Design and Back. *Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR'99)*, p. 1–8, 1999.

MEDVIDOVIC, N. et al. Using Object-oriented Typing to Support Architectural Design in the C2 Style. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 21, n. 6, p. 24–32, 1996.

MEDVIDOVIC, N. et al. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 11, n. 1, p. 2–57, 2002.

MEDVIDOVIC, N.; TAYLOR, R. N. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, v. 26, n. 1, p. 70–93, 2000.

MEI, H. et al. ABC/ADL: an ADL supporting component composition. *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods (ICFEM)*, Springer, p. 38–47, 2002.

MELO, M. d. S.; SOARES, M. S. Model-driven Structural Design of Software-intensive Systems Using SysML Blocks and UML Classes. *Proceedings of the International Conference on Enterprise Information Systems (ICEIS)*, SCITEPRESS, v. 2, p. 193–200, 2014.

MENS, T.; STRAETEN, R. V. D.; D'HONDT, M. Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. *International Conference on Model Driven Engineering Languages and Systems*, Springer, p. 200–214, 2006.

NIZ, D. D. Diagrams and Languages for Model-based Software Engineering of Embedded Systems: UML and AADL. *White Paper, www.sei.cmu.edu/library*, 2007.

OMG. OMG Systems Modeling Language (OMG SysML). *OMG document: 2015-06-03*, p. 346, 2015.

OUSSALAH, M.; SMEDA, A.; KHAMMACI, T. An Explicit Definition of Connectors for Component-based Software Architecture. *Proceedings of 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, p. 44–51, 2004.

OZKAYA, M.; KLOUKINAS, C. Are We There Yet? Analyzing Architecture Description Languages for Formal Analysis, Usability, and Realizability. *39th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, p. 177–184, 2013.

PANDEY, R. Architectural Description Languages (ADLs) vs UML: a Review. *ACM SIGSOFT Software Engineering Notes*, v. 35, n. 3, p. 1–5, 2010.

PÉREZ, J. et al. Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology. *3rd Int Conference on .NET Technologies*, .NET Technologies, 2005.

PEREZ, J. et al. A Modelling Proposal for Aspect-oriented Software Architectures. *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems(ECBS)*, p. 10–pp, 2006.

PÉREZ, J. et al. Graphical Modelling For Aspect Oriented SA. *Proceedings of the 2006 ACM symposium on Applied computing*, ACM, p. 1597–1598, 2006.

PÉREZ, J. et al. PRISMA: Towards Quality, Aspect Oriented and Dynamic Software Architectures. *Third International Conference on Quality Software*, IEEE, p. 59–66, 2003.

PÉREZ-MARTÍNEZ, J. E.; SIERRA-ALONSO, A. UML 1.4 versus UML 2.0 as Languages to Describe Software Architectures. *Software Architecture: First European Workshop, EWSA 2004*, Springer, v. 3047, p. 88, 2004.

PERRY, D. E.; WOLF, A. L. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, v. 17, n. 4, p. 40–52, 1992.

PETTY, M. D.; MCKENZIE, F. D.; XU, Q. Using a Software Architecture Description Language to Model the Architecture and Run-time Performance of a Federate. *Proceedings of the 6th IEEE International Workshop on Distributed Simulation and Real-Time Applications*, IEEE, p. 85–92, 2002.

PINTO, M.; FUENTES, L.; TROYA, J. M. DAOP-ADL: an Architecture Description Language for Dynamic Component and Aspect-based Development. *Generative Programming and Component Engineering*, p. 118–137, 2003.

PLÁŠIL, F.; BÁLEK, D.; JANEČEK, R. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. *Proceedings of Fourth International Conference on Configurable Distributed Systems*, p. 43–51, 1998.

PLASIL, F.; VISNOVSKY, S. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, v. 28, n. 11, p. 1056–1076, 2002.

RAPIDE, T. D. Draft: Guide to the Rapide 1.0 Language Reference Manuals. July, 1997.

RICCOBENE, E.; SCANDURRA, P. Integrating the SysML and the SystemC-UML Profiles in a Model-driven Embedded System Design Flow. *Design Automation for Embedded Systems*, v. 16, n. 3, p. 53–91, 2012.

ROBBINS, J. E. et al. Integrating Architecture Description Languages with a Standard Design Method. *Proceedings of the 20th International Conference on Software Engineering*, p. 209–218, 1998.

ROH, S.; KIM, K.; JEON, T. Architecture Modeling Language based on UML2. 0. *Asia-Pacific Software Engineering Conference*, p. 663–669, 2004.

ROYCE, W.; ROYCE, W. Software Architecture: Integrating Process and Technology. *TRW Quest*, v. 14, n. 1, p. 2–15, 1991.

RUCHKIN, I.; SCHMERL, B.; GARLAN, D. IPL: A Language for Model Integration Properties in Cyber-Physical Systems. Acme, Carnegie Mellon University, 2016.

RUNESON, P.; HÖST, M. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical software engineering*, v. 14, n. 2, p. 131–164, 2009.

SANGIOVANNI-VINCENTELLI, A.; NATALE, M. D. Embedded System Design for Automotive Applications. *Computer*, IEEE, v. 40, n. 10, 2007.

SANTORO, A. *Case Study in Prototyping with Rapide: Shared Memory Multiprocessor System*. California: Computer Systems Laboratory of Stanford University, 1993.

SCHOBBENS, P.-Y.; HEYMANS, P.; TRIGAUX, J.-C. Feature Diagrams: A Survey and a Formal Semantics. *14th IEEE International Conference on Requirements Engineering*, IEEE, p. 139–148, 2006.

SELIC, B. On Modeling Architectural Structures with UML. *Proc. of the Workshop on Describing Software Architecture with UML, in ICSE 2001*, 2001.

SHAW, M. The Coming-of-age of Software Architecture Research. *Proceedings of the 23rd International Conference on Software Engineering*, p. 656–664, 2001.

SHAW, M. What Makes Good Research in Software Engineering? *International Journal on Software Tools for Technology Transfer*, v. 4, n. 1, p. 1–7, 2002.

SHIRAISHI, S. Qualitative Comparison of ADL-Based Approaches to Real-World Automotive System Development. *Information and Media Technologies*, Information and Media Technologies, v. 8, n. 1, p. 196–207, 2013.

SINGH, R. International Standard ISO/IEC 12207 Software Life Cycle Processes. *Software Process Improvement and Practice*, v. 2, n. 1, p. 35–50, 1996.

SMEDA, A. et al. Cosastudio: A Software Architecture Modeling Tool. *World Academy of Science, Engineering and Technology*, v. 49, p. 263–266, 2009.

SOARES, M. S.; VRANCKEN, J. Requirements Specification and Modeling through SysML. *IEEE International Conference on Systems, Man and Cybernetics (ISIC)*, p. 1735–1740, 2007.

SOARES, M. S.; VRANCKEN, J. Model-Driven User Requirements Specification Using SysML. *Journal of Software*, v. 3, n. 6, p. 57–68, 2008.

SPAAN, R. et al. *Secure Updates in Automotive Systems*. Nijmegen: Radboud University, 2016. 1–71 p.

SUPER-INTERESSANTE. *Como Funciona o Airbag*. 2016. <http://super.abril.com.br/tecnologia/como-funciona-o-airbag>. Acesso em: 20 março 2017.

TAYLOR, R. N. et al. A Component-and Message-based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, v. 22, n. 6, p. 390–406, 1996.

TERRA, R.; VALENTE, M. T. A Dependency Constraint Language to Manage Object-oriented Software Architectures. *Software: Practice and Experience*, Wiley Online Library, v. 39, n. 12, p. 1073–1094, 2009.

TOULSON, R.; WILMSHURST, T. *Fast and Effective Embedded Systems Design: Applying the ARM mbed*. USA: Newnes, 2016.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. *Introdução à engenharia de software experimental*. Rio de Janeiro: COPPE/UFRJ, 2002. v. 1. 1–53 p.

VARONA-GOMEZ, R.; VILLAR, E. AADL Simulation and Performance Analysis in SystemC. *14th IEEE International Conference on Engineering of Complex Computer Systems*, IEEE, p. 323–328, 2009.

VLIET, H. V. Software Architecture Knowledge Management. *ASWEC 2008. 19th Australian Conference on Software Engineering*, p. 24–31, 2008.

VOGEL-HEUSER, B. et al. Usability Experiments to Evaluate UML/SysML-based Model Driven Software Engineering Notations for Logic Control in Manufacturing Automation. *Journal of Software Engineering and Applications*, v. 7, n. 11, p. 943, 2014.

WING, J. M. A Specifier's Introduction to Formal Methods. *Computer*, IEEE, v. 23, n. 9, p. 8–22, 1990.

XU, L. et al. Towards Modeling Non-functional Requirements in Software Architecture. *Early Aspects*, 2005.

ZACHMAN, J. A. The Zachman Framework: the Official Concise Definition. *Available on http://www. zachmaninternational. com*, v. 16, 2009.

ZELESNIK, G. The UniCon Language Reference Manual. *School of Computer Science Carnegie Mellon, Pittsburgh, Pennsylvania (May 1996)*, 1996.

ZURAWSKI, R. *Embedded Systems Handbook, 2-Volume Set*. Taylor and Francis Group: CRC Press, Inc, 2009.